操作系统实验报告

姓名：白思雨

学号：2186123935

日期：2020 年 12 月

实验：Tiny Shell 代码实现

1. 概述

1.1 实验任务

通过代码填空的方式实现一个简单的 linux shell，拥有执行前后台作业、查看作业信息、回收僵尸进程、暂停/恢复/终止作业等功能。

1.2 结果综述

1. 成功实现了一个 Tiny shell，通过了附带的 16 个 test 脚本测试，并具有一定鲁棒性，对于系统调用以及所有可能出错的函数都提供了 error handling，有能力应对 test 之外的其他苛刻情况。

2. 本实验所完成的 shell，输出并不遵循 shlab.pdf 中的要求——与 tshref 的输出一模一样，但也完成了相关工作。

2. 实验过程及结果

2.1 目的

The purpose of this assignment is to become more familiar with the concepts of process control and signalling。

2.2 实验内容要点

1. The prompt should be the string “tsh> ”

这一点 tsh.c 中已经实现了，并且提供了开关 prompt 和 verbose 的一些 api。不过这些都是测试所使用的，和完成实验关系不大。

2. 分析 command line，如果输入的是内置指令，则立即执行，如果不是，则当成是一个可执行文件的路径，并 fork 一个子进程，执行该可执行文件

这一点在 tshref（实验所给的参考 shell 程序）中输入一个非法文件路径，会被当成一个未知指令，而不是一个非法的文件路径。而在我的 tinyshell 中，这一点被实现了。

<1>parse()函数

tsh.c 中已经写好了一个 parse 函数，这个函数会忽略所有前导 space，将 command line 根据空格分割，构造 argv 参数列表，并将 argv 的最后一个元素置为 NULL（为了符合 execv 的参数要求）。parse()同时还提供了单引号的忽略功能。总体来说不是很鲁棒，但也够用了。需要注意的是该函数申请了一个静态 char 数组 array，用于临时存储 command line。因为在 parse()的实现里，argv 是指向 array，所以 array 必须是静态的，否则当 parse()结束之后，argv 不但会失去作用，而且会成为野指针。

<2>其次，在 eval 函数中实现小标题所提出的功能
使用 parse()分析 command line 得到相应信息：
（1）若是空指令或者非正常指令，则直接 return；
（2）若是 built in command，则调用 builtin_cmd()，该函数的具体实现逻辑无需多讲，需要注意的是：在所有对 joblist 有写操作的语句两侧都要 mask 掉所有 signal 再恢复，防止 Races。
（3）若以上均不是，则将 argv 的第一个元素当做一个可执行文件，再 fork()出一个子程序（在 fork()前需要 mask 掉 sigchild 信号），再使用 evecv()执行。同样，父进程中后续的 job 添加等语句也需要 mask 掉所有 signal。最后，根据 FG/BG 信息判断是否要执行 waitfg()。

<3>虽然实验中没有明文要求，但显然对于 BG、FG、ST 三种状态的相互转换的情况都要考虑到：
（1）BG->FG：通过 fg 容易实现，解析参数，错误判断，安全地取出 job 和修改状态，通过 kill 发送 SIGCONT 信号（如果进程已经在 running，则默认不会对 SIGCONT 做出反应），调用 waitfg()。
（2）FG->ST：在 sigchild handler 中通过解析子进程暂停状态实现，下文详细介绍。
（3）ST->FG：通过 fg 容易实现，解析参数，错误判断，安全地取出 job 和修改状态，通过 kill 发送 SIGCONT 信号，调用 waitfg()。
（4）ST->BG：通过 bg 容易实现，解析参数，错误判断，安全地取出 job 和修改状态，通过 kill 发送 SIGCONT 信号。
（5）BG->ST：用户无法发出此类命令，因为 shell 不会向 background job 发送 SIGTSTP 信号。
（6）FG->BG：用户无法发出此类命令，因为 shell 在有 foreground job running 时不会执行 bg 指令。但是由于标准输入流的缓冲和进程之间是独立并发运作的，所以实际上在 foreground job running 时输入的指令仍然会被存入缓冲区，并且在下一次 shell 的主循环执行时会被当做 command line 读入，这一特色对于各种批处理脚本（包括这次作业的 test 脚本）来说都是无碍甚至必要的，但是对于个别用户来说可能会造成困扰。我在 sigchild handler 的相应位置加入了一些代码（默认是被注释掉的，不启动该功能），使得每次 foreground job 结束时，都会先清空 stdin 缓冲，再读入新指令，以满足某些用户的特定需求。

<4>waitfg()的实现
在 shell 内定义全局变量 waiting，代表 shell 是否正在等待一个前台进程结束\暂停，并在 sigchild handler 中对于当前 foreground job 子进程的结束作特殊判断，以决定是否改变 waiting 的值。
具体的实现遵照 pdf 的 hint，在 waitfg()通过 while 循环加上 sleep 的方式实现等待，暂不考虑性能优化。
另外，在 while 循环前后将 SIGCHILD 信号的 mask 取消、恢复，可以增强 waitfg()的鲁棒性，避免死锁。

<5> execv 的参数构造
虽然最终的结果是不需要做额外的更改，但是思考是必要的。

```
int execv(const char *pathname, char *const argv[])
```

首先，evecv()不返回（这无关紧要）。至于 evecv()参数，需要 shell 事先准备好要传入这两个参数的变量，然后它们会被 fork()复制一份，最后传入 execv()。但这两个参数都是指针，于是有以下问题：
（1）如何保证指针使用的安全性
传入 execv()的指针指向的数据会被操作系统以特定方式（分别以'\0'和 NULL 为结尾）复制一份，并新申请相应的内存空间，且指针指向的内存是被复制的那一份所申请的新内存，所以不需要担心内存泄漏或者是野指针。
（2）如何保证指针取出的数据是正确的
同理，虽然 parse()返回的 argv 指向的是静态变量 array，且该变量会在 shell 下一次调用 parse()的时候被改变，但是 argv 在传入 execv()之后指向的已经不是 shell 进程 array 了，所以没有关系。
（3）在 fork()之后，子进程执行 execv()之前，主进程可能就执行了下一次 parse()，静态变量 array 的值可

能会被改变，从而 evecv()复制时会得到错误的参数？

在 fork()执行之后，主进程 shell 的静态变量 array，以及指向它的指针 argv 都会被子进程完全独立地复制一份，且指针的取值是根据进程重定位的。所以在 fork()之后，即使 shell 第二次执行了 parse()，也只会改变 shell 本身的静态变量 array，不会影响子进程的 array。

3. ctrl-c(ctrl-z)会向 foreground job 及所有其衍生进程发送 SIGINT(SIGTSTP)信号，若没有前台进程，则没有影响。

该部分的实现逻辑也很繁复，使用 signalmask 保证对于 joblist 的安全存取就不再赘述，主要说两个重点：

<1>使用进程组实现 signal 的正确广播

这一点在 pdf 中也被反复提及。Shell 需要将 SIGINT 和 SIGTSTP 发送到 foreground job 及其所有衍生进程，能做到这一点的就是 kill 的进程组广播用法，以及 fork()“子进程和父进程同属一个进程组”的功能的结合。

在 shell 的实现中，每个 job 都要有不同的进程组 id，且要和 shell 进程的进程组 id 不同，原因有二：

（1）防止发送给 foreground job 及其衍生进程的 signal 被广播给其他 job 或者 shell 本身。

（2）发送给 shell 的 signal 被直接广播给 jobs。

发送操作自然是通过 sigint handler 和 sigtstp handler，就是简单的 getjobpid()、kill()等操作。

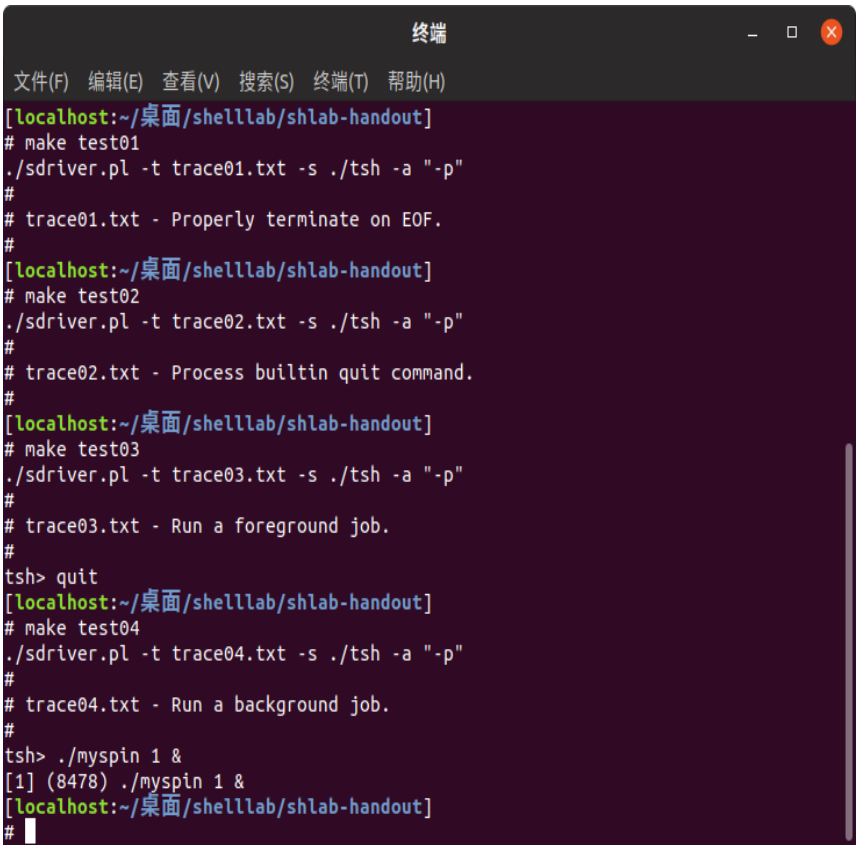
需要注意的一点：

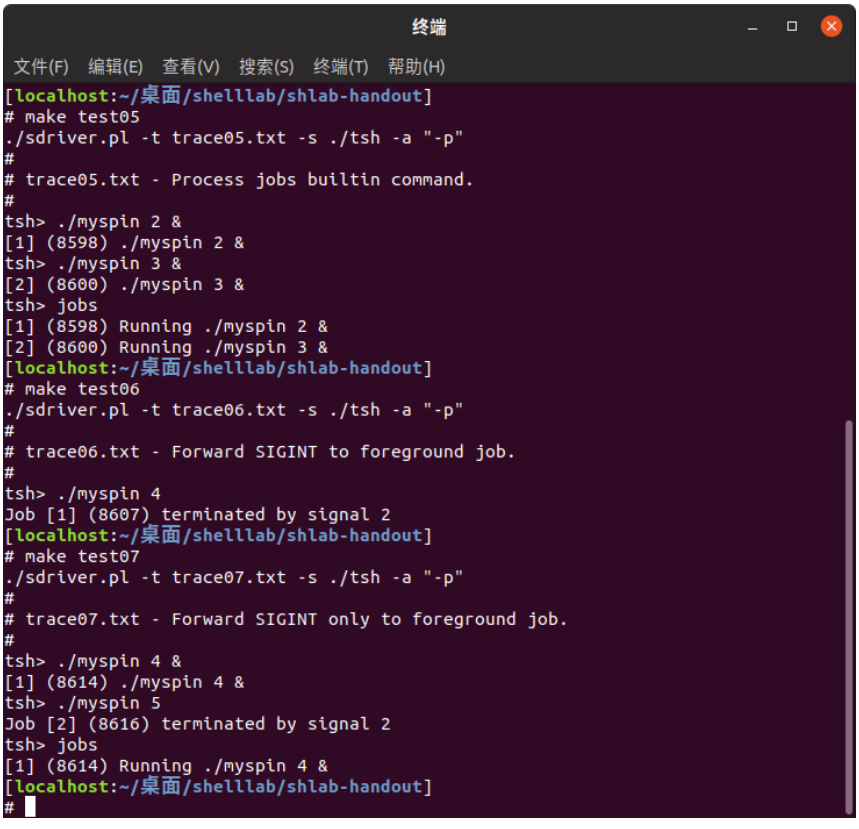
在 eval()中 fork()子进程后，调用 setpgid()，使 shell 设置其子进程的进程组 ID，然后使子进程设置其自己的进程组 ID。这些调用中有一个是冗余的，但这样做可以保证父、子进程在进一步操作之前，子进程都进入了该进程组。否则依赖于哪一个进程先执行，就产生一个竞态条件。

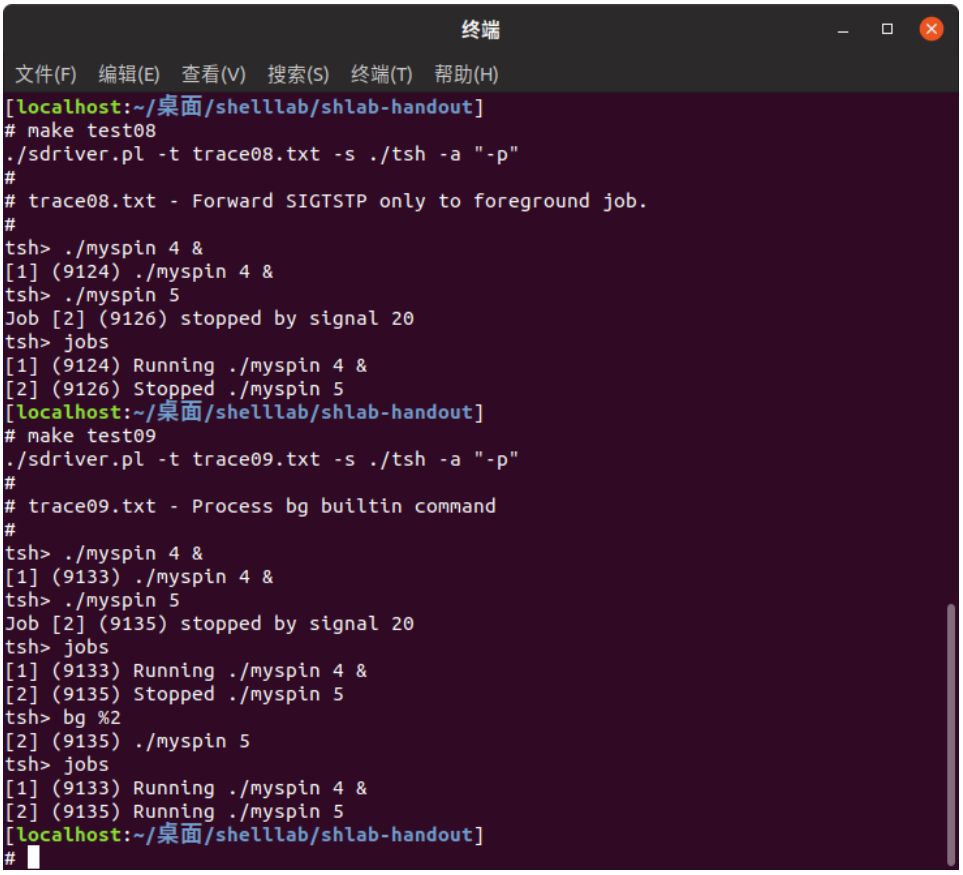
<2>使用 waitpid()取得子进程信息

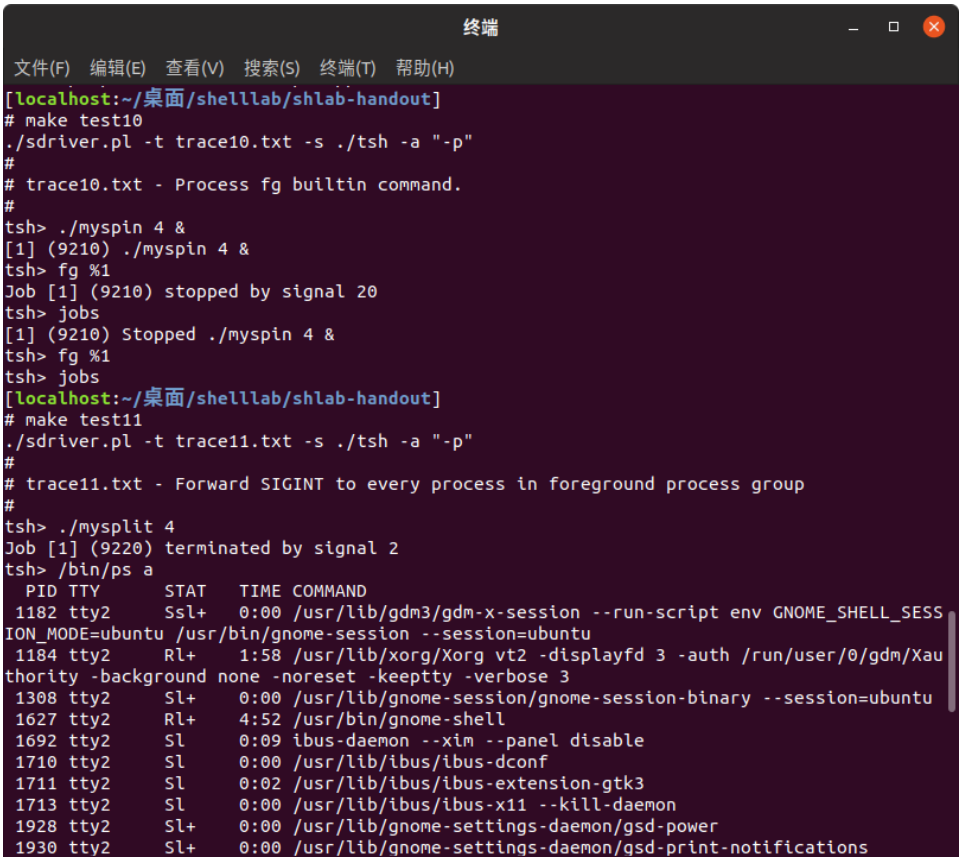
waitpid()不但可以非阻塞地等待子进程结束\暂停，还能通过 status 参数获得子进程的结束信息。通过一些宏可以方便地解析 status，以实现在 sigchild handler 中根据子进程的结束\暂停原因，来对 joblist 做更改、输出反馈信息。具体实现见代码，在此不列举。

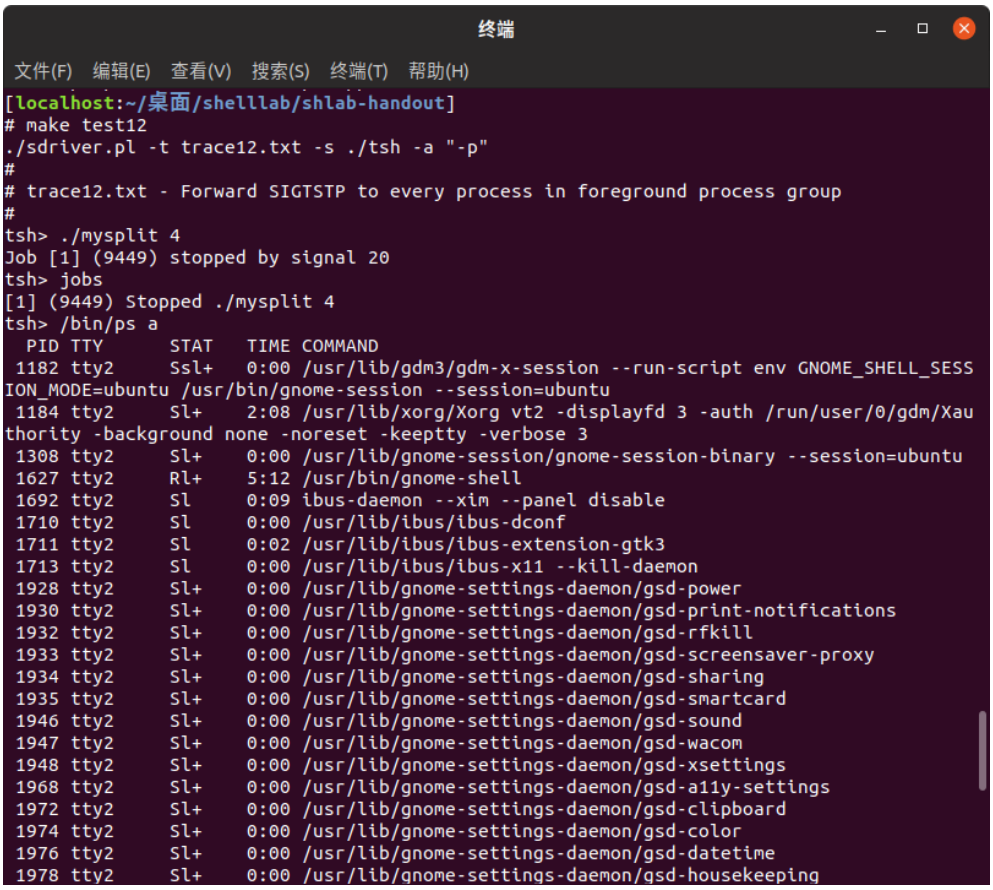
2.3 实验结果描述

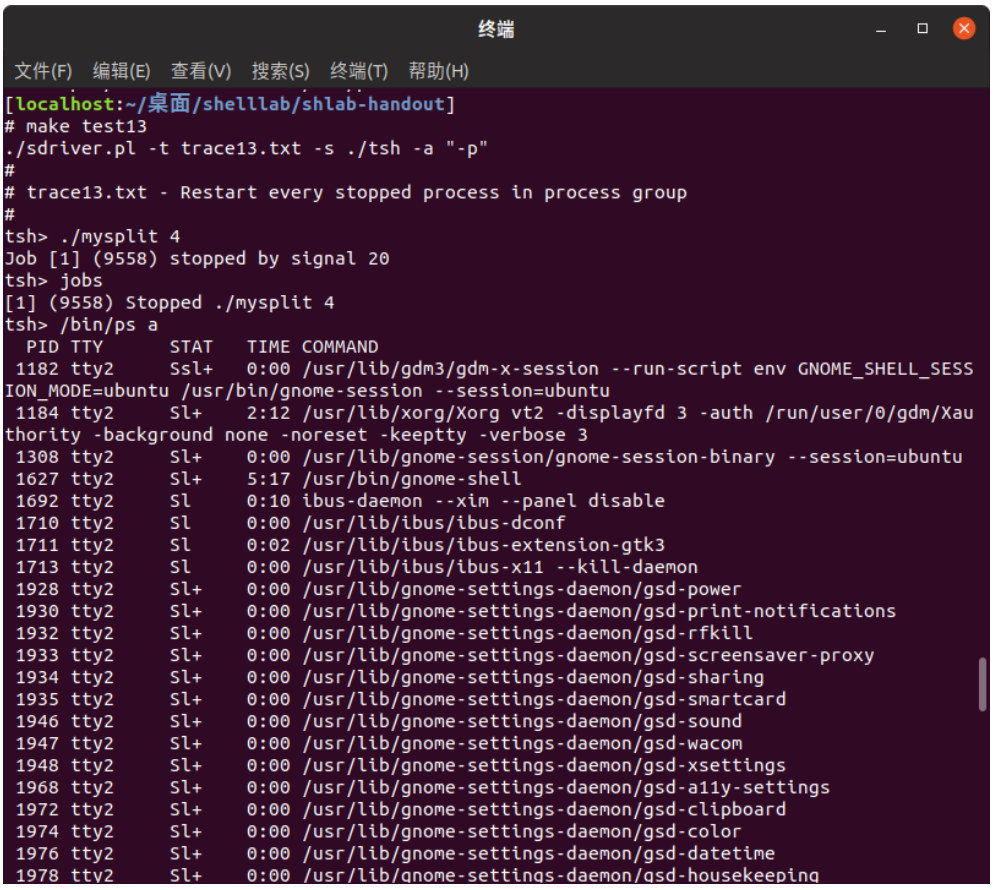

终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[localhost:~/桌面/shelllab/shlab-handout]
make test01
./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
#
trace01.txt - Properly terminate on EOF.
#
[localhost:~/桌面/shelllab/shlab-handout]
make test02
./sdriver.pl -t trace02.txt -s ./tsh -a "-p"
#
trace02.txt - Process builtin quit command.
#
[localhost:~/桌面/shelllab/shlab-handout]
make test03
./sdriver.pl -t trace03.txt -s ./tsh -a "-p"
#
trace03.txt - Run a foreground job.
#
tsh> quit
[localhost:~/桌面/shelllab/shlab-handout]
make test04
./sdriver.pl -t trace04.txt -s ./tsh -a "-p"
#
trace04.txt - Run a background job.
#
tsh> ./myspin 1 &
[1] (8478) ./myspin 1 &
[localhost:~/桌面/shelllab/shlab-handout]
#

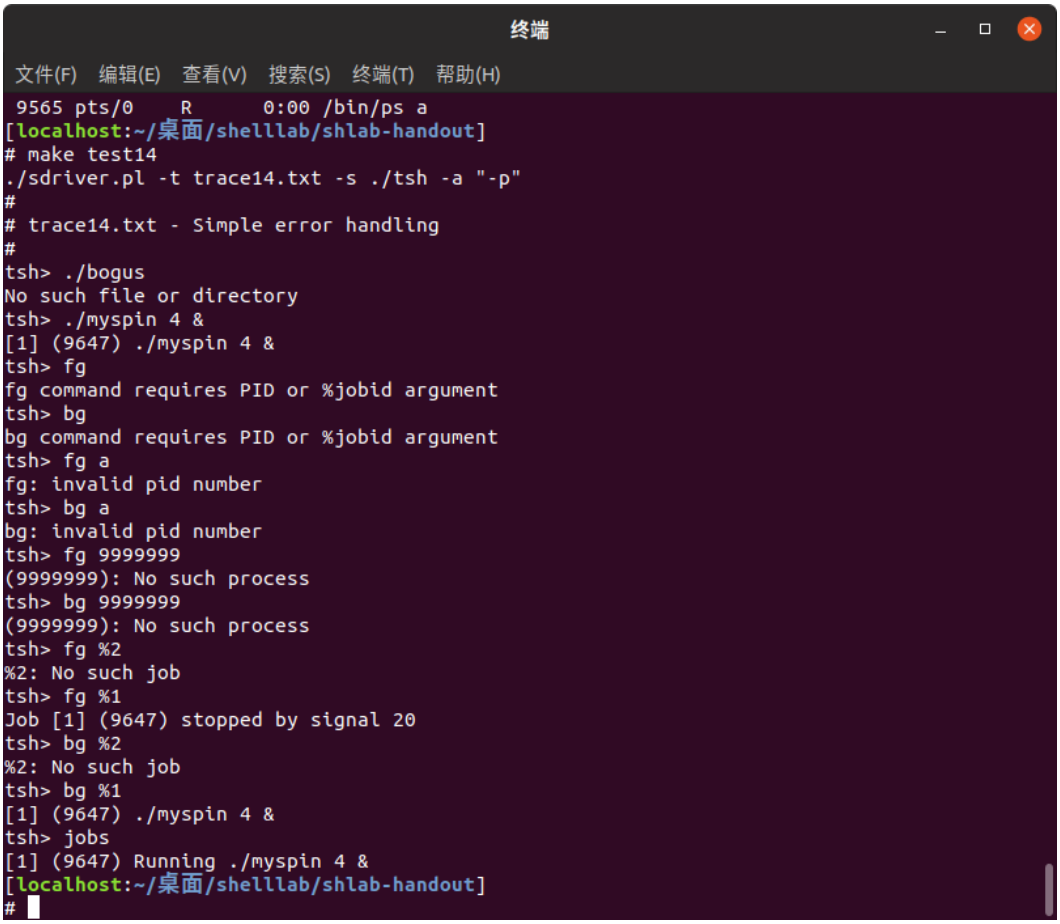

终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[localhost:~/桌面/shelllab/shlab-handout]
make test05
./sdriver.pl -t trace05.txt -s ./tsh -a "-p"
#
trace05.txt - Process jobs builtin command.
#
tsh> ./myspin 2 &
[1] (8598) ./myspin 2 &
tsh> ./myspin 3 &
[2] (8600) ./myspin 3 &
tsh> jobs
[1] (8598) Running ./myspin 2 &
[2] (8600) Running ./myspin 3 &
[localhost:~/桌面/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] (8607) terminated by signal 2
[localhost:~/桌面/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] (8614) ./myspin 4 &
tsh> ./myspin 5
Job [2] (8616) terminated by signal 2
tsh> jobs
[1] (8614) Running ./myspin 4 &
[localhost:~/桌面/shelllab/shlab-handout]
#

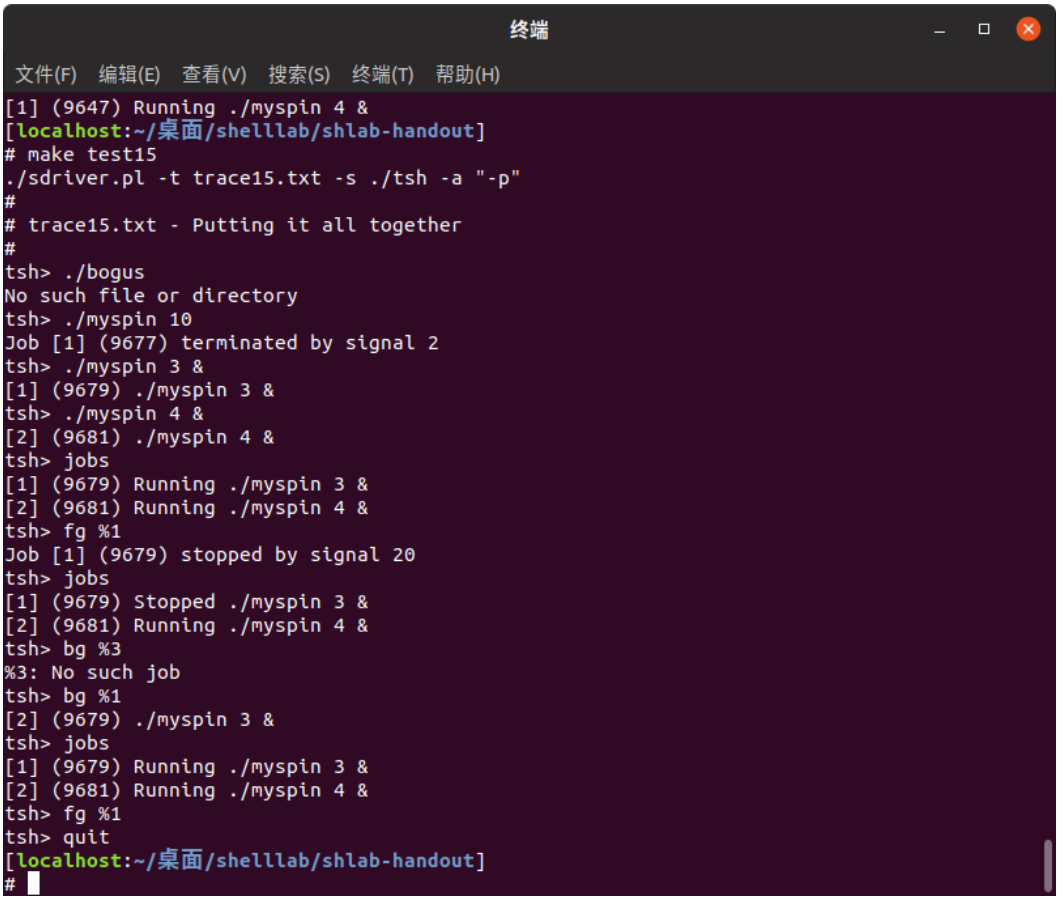

终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[localhost:~/桌面/shelllab/shlab-handout]
make test08
./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
#
trace08.txt - Forward SIGTSTP only to foreground job.
#
tsh> ./myspin 4 &
[1] (9124) ./myspin 4 &
tsh> ./myspin 5
Job [2] (9126) stopped by signal 20
tsh> jobs
[1] (9124) Running ./myspin 4 &
[2] (9126) Stopped ./myspin 5
[localhost:~/桌面/shelllab/shlab-handout]
make test09
./sdriver.pl -t trace09.txt -s ./tsh -a "-p"
#
trace09.txt - Process bg builtin command
#
tsh> ./myspin 4 &
[1] (9133) ./myspin 4 &
tsh> ./myspin 5
Job [2] (9135) stopped by signal 20
tsh> jobs
[1] (9133) Running ./myspin 4 &
[2] (9135) Stopped ./myspin 5
tsh> bg %2
[2] (9135) ./myspin 5
tsh> jobs
[1] (9133) Running ./myspin 4 &
[2] (9135) Running ./myspin 5
[localhost:~/桌面/shelllab/shlab-handout]
#


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[localhost:~/桌面/shelllab/shlab-handout]
make test10
./sdriver.pl -t trace10.txt -s ./tsh -a "-p"
#
trace10.txt - Process fg builtin command.
#
tsh> ./myspin 4 &
[1] (9210) ./myspin 4 &
tsh> fg %1
Job [1] (9210) stopped by signal 20
tsh> jobs
[1] (9210) Stopped ./myspin 4 &
tsh> fg %1
tsh> jobs
[localhost:~/桌面/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] (9220) terminated by signal 2
tsh> /bin/ps a
PID TTY STAT TIME COMMAND
1182 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESS
ION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
1184 tty2 Rl+ 1:58 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/0/gdm/Xau
thority -background none -noreset -keeptty -verbose 3
1308 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu
1627 tty2 Rl+ 4:52 /usr/bin/gnome-shell
1692 tty2 Sl 0:09 ibus-daemon --xim --panel disable
1710 tty2 Sl 0:00 /usr/lib/ibus/ibus-dconf
1711 tty2 Sl 0:02 /usr/lib/ibus/ibus-extension-gtk3
1713 tty2 Sl 0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1928 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-power
1930 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[localhost:~/桌面/shelllab/shlab-handout]
make test12
./sdriver.pl -t trace12.txt -s ./tsh -a "-p"
#
trace12.txt - Forward SIGTSTP to every process in foreground process group
#
tsh> ./mysplit 4
Job [1] (9449) stopped by signal 20
tsh> jobs
[1] (9449) Stopped ./mysplit 4
tsh> /bin/ps a
PID TTY STAT TIME COMMAND
1182 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESS
ION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
1184 tty2 Sl+ 2:08 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/0/gdm/Xau
thority -background none -noreset -keeptty -verbose 3
1308 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu
1627 tty2 Rl+ 5:12 /usr/bin/gnome-shell
1692 tty2 Sl 0:09 ibus-daemon --xim --panel disable
1710 tty2 Sl 0:00 /usr/lib/ibus/ibus-dconf
1711 tty2 Sl 0:02 /usr/lib/ibus/ibus-extension-gtk3
1713 tty2 Sl 0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1928 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-power
1930 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1932 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1933 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1934 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1935 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1946 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-sound
1947 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-wacom
1948 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1968 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1972 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1974 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-color
1976 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1978 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[localhost:~/桌面/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] (9558) stopped by signal 20
tsh> jobs
[1] (9558) Stopped ./mysplit 4
tsh> /bin/ps a
PID TTY STAT TIME COMMAND
1182 tty2 Ssl+ 0:00 /usr/lib/gdm3/gdm-x-session --run-script env GNOME_SHELL_SESS
ION_MODE=ubuntu /usr/bin/gnome-session --session=ubuntu
1184 tty2 Sl+ 2:12 /usr/lib/xorg/Xorg vt2 -displayfd 3 -auth /run/user/0/gdm/Xau
thority -background none -noreset -keeptty -verbose 3
1308 tty2 Sl+ 0:00 /usr/lib/gnome-session/gnome-session-binary --session=ubuntu
1627 tty2 Sl+ 5:17 /usr/bin/gnome-shell
1692 tty2 Sl 0:10 ibus-daemon --xim --panel disable
1710 tty2 Sl 0:00 /usr/lib/ibus/ibus-dconf
1711 tty2 Sl 0:02 /usr/lib/ibus/ibus-extension-gtk3
1713 tty2 Sl 0:00 /usr/lib/ibus/ibus-x11 --kill-daemon
1928 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-power
1930 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-print-notifications
1932 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-rfkill
1933 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-screensaver-proxy
1934 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-sharing
1935 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-smartcard
1946 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-sound
1947 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-wacom
1948 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-xsettings
1968 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-a11y-settings
1972 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-clipboard
1974 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-color
1976 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-datetime
1978 tty2 Sl+ 0:00 /usr/lib/gnome-settings-daemon/gsd-housekeeping


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
9565 pts/0 R 0:00 /bin/ps a
[localhost:~/桌面/shelllab/shlab-handout]
make test14
./sdriver.pl -t trace14.txt -s ./tsh -a "-p"
#
trace14.txt - Simple error handling
#
tsh> ./bogus
No such file or directory
tsh> ./myspin 4 &
[1] (9647) ./myspin 4 &
tsh> fg
fg command requires PID or %jobid argument
tsh> bg
bg command requires PID or %jobid argument
tsh> fg a
fg: invalid pid number
tsh> bg a
bg: invalid pid number
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] (9647) stopped by signal 20
tsh> bg %2
%2: No such job
tsh> bg %1
[1] (9647) ./myspin 4 &
tsh> jobs
[1] (9647) Running ./myspin 4 &
[localhost:~/桌面/shelllab/shlab-handout]
#


终端
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[1] (9647) Running ./myspin 4 &
[localhost:~/桌面/shelllab/shlab-handout]
make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
#
trace15.txt - Putting it all together
#
tsh> ./bogus
No such file or directory
tsh> ./myspin 10
Job [1] (9677) terminated by signal 2
tsh> ./myspin 3 &
[1] (9679) ./myspin 3 &
tsh> ./myspin 4 &
[2] (9681) ./myspin 4 &
tsh> jobs
[1] (9679) Running ./myspin 3 &
[2] (9681) Running ./myspin 4 &
tsh> fg %1
Job [1] (9679) stopped by signal 20
tsh> jobs
[1] (9679) Stopped ./myspin 3 &
[2] (9681) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[2] (9679) ./myspin 3 &
tsh> jobs
[1] (9679) Running ./myspin 3 &
[2] (9681) Running ./myspin 4 &
tsh> fg %1
tsh> quit
[localhost:~/桌面/shelllab/shlab-handout]
#

Test 脚本测试结果：16 个 test 测试结果均和 tshref 的结果相同（反馈信息的句式不一定完全相同）。

如同 shlab.pdf 所说，评判本次实验结果的标准不止是 test。我 tinyshell 代码中也添加有完善的注释（英文），且对于系统调用和其他函数的调用都做了 error handling。另外，对于这 16 个 test 所没有测试到的条件也都进行了实现和测试（例如子进程因为除 sigint、sigtstp 的其他 signal 而结束、暂停、子进程意外结束），没有出 bug，证明 tinyshell 鲁棒性良好。

3. 总结体会与建议

操作系统课程的后半部分难度比较大，但是也更有趣一些。学到了很多东西，也在实验中体会到了计算机系统的趣味性，真的是很棒的体验。其实我并不是很常用 unix 系统，但由于这个实验我打开了大一装好但没打开几次的 Ubuntu 系统。

要说建议的话，希望老师注意身体，老师好像经常生病，一定要好好休息，这样才能给后面的同学带来更多这样好的课程体验。

4. 实验代码

```c
/*
 * tsh - A tiny shell program with job control
 *
 * Baisiyu Bsyess
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
```

```c
#include <signal.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

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

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

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

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

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

```
55.
56.
57. /* Function prototypes */
58.
59. /* Here are the functions that you will implement */
60. void eval(char *cmdline);
61. int builtin_cmd(char **argv);
62. void do_bgfg(char **argv);
63. void waitfg(pid_t pid);
64.
65. void sigchld_handler(int sig);
66. void sigtstp_handler(int sig);
67. void sigint_handler(int sig);
68.
69. /* Here are helper routines that we've provided for you */
70. int parseline(const char *cmdline, char **argv);
71. void sigquit_handler(int sig);
72.
73. void clearjob(struct job_t *job);
74. void initjobs(struct job_t *jobs);
75. int maxjid(struct job_t *jobs);
76. int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline);
77. int deletejob(struct job_t *jobs, pid_t pid);
78. pid_t fgpid(struct job_t *jobs);
79. struct job_t *getjobpid(struct job_t *jobs, pid_t pid);
80. struct job_t *getjobjid(struct job_t *jobs, int jid);
81. int pid2jid(pid_t pid);
82. void listjobs(struct job_t *jobs);
83. int str2int(char *str); //parse a string to positive integer
84.
85. void usage(void);
86. void unix_error(char *msg);
87. void app_error(char *msg);
88. typedef void handler_t(int);
89. handler_t *Signal(int signum, handler_t *handler);
90.
91. /*
92.  * main - The shell's main routine
93.  */
94. int main(int argc, char **argv)
95. {
96.     char c;
97.     char cmdline[MAXLINE];
98.     int emit_prompt = 1; /* emit prompt (default) */
```

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

```
143.         app_error("fgets error");
144.     if (feof(stdin)) { /* End of file (ctrl-d) */
145.         fflush(stdout);
146.         exit(0);
147.     }
148. 
149.     /* Evaluate the command line */
150.     eval(cmdline);
151.     fflush(stdout);
152.     fflush(stdout);
153.     }
154. 
155.     exit(0); /* control never reaches here */
156. }
157. 
158. /*
159.  * eval - Evaluate the command line that the user has just typed in
160.  *
161.  * If the user has requested a built-in command (quit, jobs, bg or fg)
162.  * then execute it immediately. Otherwise, fork a child process and
163.  * run the job in the context of the child. If the job is running in
164.  * the foreground, wait for it to terminate and then return.  Note:
165.  * each child process must have a unique process group ID so that our
166.  * background children don't receive SIGINT (SIGTSTP) from the kernel
167.  * when we type ctrl-c (ctrl-z) at the keyboard.
168. */
169. void eval(char *cmdline)
170. {
171.     char* argv[MAXLINE];
172.     //parse the command line
173.     int if_BG = parseline(cmdline, argv);
174.     //if its a blank or abnormal command
175.     if(argv[0] == NULL)
176.         return;
177.     //if its a build-in command
178.     if(builtin_cmd(argv))
179.         return;
180.     //if it wants to run something
181.     sigset_t mask_all, mask_one, prev_one;
182.     sigfillset(&mask_all);
183.     sigemptyset(&mask_one);
184.     sigaddset(&mask_one, SIGCHLD);
185. 
186.     sigprocmask(SIG_BLOCK, &mask_one, &prev_one);
```

```
187.     pid_t child_pid = fork();
188.     if(child_pid==0)
189.     {//child progress
190.         setpgid(0, 0);//set child progress as the leader of the progress group
191.         sigprocmask(SIG_SETMASK, &prev_one, NULL);
192.         if(execv(argv[0], argv) ==  -1)
193.         {
194.             perror("");//execv error
195.             exit(1);
196.         }
197.     }
198.     else
199.     {//parent progress
200.         sigprocmask(SIG_BLOCK, &mask_all, NULL);
201.         setpgid(child_pid, child_pid);//set child progress as the leader of the progress
    group, for sure
202.         if(if_BG)//new BG job
203.         {
204.             if(!addjob(jobs, child_pid, BG, cmdline))//if joblist is full
205.             {
206.                 kill(-child_pid, SIGKILL);
207.                 if(errno==ESRCH)
208.                 {
209.                     perror("kill failed");
210.                     exit(1);
211.                 }
212.             }
213.             printf("[%d] (%d) %s", nextjid==1?MAXJID:(nextjid-
    1), child_pid, cmdline);//print the new bg job info
214.             fflush(stdout);
215.             sigprocmask(SIG_SETMASK, &prev_one, NULL);
216.
217.         }
218.         else//new FG job
219.         {
220.             if(!addjob(jobs, child_pid, FG, cmdline))//if joblist is full
221.             {
222.                 kill(-child_pid, SIGKILL);
223.                 if(errno==ESRCH)
224.                 {
225.                     perror("kill failed");
226.                     exit(1);
227.                 }
228.             }
```

```
229.             sigprocmask(SIG_SETMASK, &prev_one, NULL);
230.             waitfg(child_pid);
231.         }
232.         return;
233.     }
234.     return;//never reach here
235.
236. }
237.
238. /*
239.  * parseline - Parse the command line and build the argv array.
240.  *
241.  * Characters enclosed in single quotes are treated as a single
242.  * argument.  Return true if the user has requested a BG job, false if
243.  * the user has requested a FG job.
244.  */
245. int parseline(const char *cmdline, char **argv)
246. {
247.     static char array[MAXLINE]; /* holds local copy of command line */
248.     char *buf = array;          /* pointer that traverses command line */
249.     char *delim;                /* points to first space delimiter */
250.     int argc;                   /* number of args */
251.     int bg;                     /* background job? */
252.
253.     strcpy(buf, cmdline);
254.     buf[strlen(buf)-1] = ' ';  /* replace trailing '\n' with space */
255.     while (*buf && (*buf == ' ')) /* ignore leading spaces */
256.     buf++;
257.
258.     /* Build the argv list */
259.     argc = 0;
260.     if (*buf == '\'') {
261.     buf++;
262.     delim = strchr(buf, '\'');
263.     }
264.     else {
265.     delim = strchr(buf, ' ');
266.     }
267.
268.     while (delim) {
269.     argv[argc++] = buf;
270.     *delim = '\0';
271.     buf = delim + 1;
272.     while (*buf && (*buf == ' ')) /* ignore spaces */
```

```
273.            buf++;
274.
275.    if (*buf == '\'') {
276.        buf++;
277.        delim = strchr(buf, '\'');
278.    }
279.    else {
280.        delim = strchr(buf, ' ');
281.    }
282.    }
283.    argv[argc] = NULL;
284.
285.    if (argc == 0)  /* ignore blank line */
286.    return 1;
287.
288.    /* should the job run in the background? */
289.    if ((bg = (*argv[argc-1] == '&')) != 0) {
290.    argv[--argc] = NULL;
291.    }
292.    return bg;
293. }
294.
295. /*
296.  * builtin_cmd - If the user has typed a built-in command then execute
297.  *    it immediately.
298.  */
299. int builtin_cmd(char **argv)
300. {
301.    sigset_t mask_all, prev_one;
302.    sigfillset(&mask_all);
303.    if(!strcmp(argv[0], "jobs"))//the "jobs" command
304.    {
305.        listjobs(jobs);
306.        return 1;
307.    }
308.    else if(!strcmp(argv[0], "quit"))//the "quit" command
309.    {
310.        exit(0);
311.    }
312.    else if(!strcmp(argv[0], "bg")||!strcmp(argv[0], "fg"))//the "bg" or "fg" command
313.    {
314.        do_bgfg(argv);
315.    return 1;
316.    }
```

```
317.     return 0;     /* not a builtin command */
318. }
319.
320. /*
321.  * do_bgfg - Execute the builtin bg and fg commands
322.  */
323. void do_bgfg(char **argv)
324. {
325.     sigset_t mask_all, prev_one;
326.     if(!strcmp(argv[0], "bg"))//the "bg" command
327.     {
328.         if(argv[1]==NULL)//no argument
329.         {
330.             printf("bg command requires PID or %%jobid argument\n");
331.             fflush(stdout);
332.             return;
333.         }
334.         if(argv[1][0]=='%')//jid input
335.         {
336.             int jid;
337.             if((jid=str2int(argv[1]+1))==-1)//invalid argument
338.             {
339.                 printf("bg: invalid jid number\n");
340.                 fflush(stdout);
341.                 return;
342.             }
343.
344.             //do sigmask in case that signal handler may delete the job
345.             //just after we get the job
346.             sigprocmask(SIG_SETMASK, &mask_all, &prev_one);
347.             struct job_t *job = getjobjid(jobs, jid);
348.             if(job==NULL)
349.             {
350.                 printf("%%%d: No such job\n",jid);
351.                 fflush(stdout);
352.                 sigprocmask(SIG_SETMASK, &prev_one, NULL);
353.                 return;
354.             }
355.             job->state = BG;//set the job state
356.             printf("[%d] (%d) %s", nextjid==1?MAXJID:(nextjid-
    1), job->pid, job->cmdline);//print the new bg job info
357.             fflush(stdout);
358.             sigprocmask(SIG_SETMASK, &prev_one, NULL);
359.
```

```
360.             kill(-(job->pid), SIGCONT);//send SIGCONT signal
361.             if(errno==ESRCH)
362.             {
363.                 perror("kill failed");
364.                 exit(1);
365.             }
366.
367.         }
368.         else//pid input
369.         {
370.             pid_t pid;
371.             if((pid=str2int(argv[1]))==-1)//invalid argument
372.             {
373.                 printf("bg: invalid pid number\n");
374.                 fflush(stdout);
375.                 return;
376.             }
377.
378.             //do sigmask in case that signal handler may delete the job
379.             //just after we get the job
380.             sigprocmask(SIG_SETMASK, &mask_all, &prev_one);
381.             struct job_t *job = getjobpid(jobs, pid);
382.             if(job==NULL)
383.             {
384.                 printf("(%d): No such process\n",pid);
385.                 fflush(stdout);
386.                 sigprocmask(SIG_SETMASK, &prev_one, NULL);
387.                 return;
388.             }
389.             job->state = BG;
390.             printf("[%d] (%d) %s", nextjid==1?MAXJID:(nextjid-
    1), job->pid, job->cmdline);//print the new bg job info
391.             fflush(stdout);
392.             sigprocmask(SIG_SETMASK, &prev_one, NULL);
393.
394.             kill(-(job->pid), SIGCONT);
395.             if(errno==ESRCH)
396.             {
397.                 perror("kill failed");
398.                 exit(1);
399.             }
400.         }
401.         return;
402.
```

```
403.     }
404.     else //the "fg" command
405.     {
406.         if(argv[1]==NULL)//no argument
407.         {
408.             printf("fg command requires PID or %%jobid argument\n");
409.             fflush(stdout);
410.             return;
411.         }
412.         if(argv[1][0]=='%')//jid input
413.         {
414.             int jid;
415.             if((jid=str2int(argv[1]+1))==-1)//invalid argument
416.             {
417.                 printf("fg: invalid jid number\n");
418.                 fflush(stdout);
419.                 return;
420.             }
421. 
422.             //do sigmask in case that signal handler may delete the job
423.             //just after we get the job
424.             sigprocmask(SIG_SETMASK, &mask_all, &prev_one);
425.             struct job_t *job = getjobjid(jobs, jid);
426.             if(job==NULL)
427.             {
428.                 printf("%%%d: No such job\n",jid);
429.                 fflush(stdout);
430.                 sigprocmask(SIG_SETMASK, &prev_one, NULL);
431.                 return;
432.             }
433.             job->state = FG;
434.             sigprocmask(SIG_SETMASK, &prev_one, NULL);
435. 
436.             kill(-(job->pid), SIGCONT);
437.             if(errno==ESRCH)
438.             {
439.                 perror("kill failed");
440.                 exit(1);
441.             }
442.             waitfg(job->pid);//wait for the foreground job's termination
443.         }
444.         else//pid input
445.         {
446.             pid_t pid;
```

```
447.             if((pid=str2int(argv[1]))==-1)//invalid argument
448.             {
449.                 printf("fg: invalid pid number\n");
450.                 fflush(stdout);
451.                 return;
452.             }
453.
454.             //do sigmask in case that signal handler may delete the job
455.             //just after we get the job
456.             sigprocmask(SIG_SETMASK, &mask_all, &prev_one);
457.             struct job_t *job = getjobpid(jobs, pid);
458.             if(job==NULL)
459.             {
460.                 printf("(%d): No such process\n",pid);
461.                 fflush(stdout);
462.                 sigprocmask(SIG_SETMASK, &prev_one, NULL);
463.                 return;
464.             }
465.             job->state = FG;
466.             sigprocmask(SIG_SETMASK, &prev_one, NULL);
467.
468.             kill(-(job->pid), SIGCONT);
469.             if(errno==ESRCH)
470.             {
471.                 perror("kill failed");
472.                 exit(1);
473.             }
474.             waitfg(job->pid);//wait for the foreground job's termination
475.         }
476.         return;
477.      }
478.
479.
480.     return;
481. }
482.
483.
484. /*
485.  * waitfg - Block until process pid is no longer the foreground process
486.  */
487. void waitfg(pid_t pid)
488. {
489.     sigset_t no_mask, prev_one;
490.     sigemptyset(&no_mask);
```

```
491.     //set sigmask empty to prevent infinite waiting
492.     sigprocmask(SIG_SETMASK, &no_mask, &prev_one);
493.     //Global variable waiting, infer that if the shell is waiting for a fg job
494.     waiting = true;
495.     while(waiting && getjobpid(jobs,pid)->state==FG)
496.     {
497.         sleep(0.1);
498.     }
499.     sigprocmask(SIG_SETMASK, &prev_one, NULL);
500.     return;
501. }
502. 
503. /*****************
504.  * Signal handlers
505.  *****************/
506. 
507. /*
508.  * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
509.  *     a child job terminates (becomes a zombie), or stops because it
510.  *     received a SIGSTOP or SIGTSTP signal. The handler reaps all
511.  *     available zombie children, but doesn't wait for any other
512.  *     currently running children to terminate.
513.  */
514. void sigchld_handler(int sig)
515. {
516.     sigset_t mask_all, prev_all;
517.     pid_t pidfg, pid;
518.     int status = -1;
519.     sigfillset(&mask_all);
520.     if((pidfg=fgpid(jobs)) != 0)//if there is a foreground job running
521.     {
522. 
523.         int res = waitpid(pidfg, &status, WNOHANG|WUNTRACED);
524.         if(res != 0)//if the terminated child is the fg job
525.         {
526.             if(WIFEXITED(status))//if the child is terminated by exit
527.             {
528.                 sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
529.                 deletejob(jobs, pidfg);
530.                 waiting = false;//stop waiting for fg job
531.                 sigprocmask(SIG_SETMASK, &prev_all, NULL);
532.             }
533. 
534.             else if(WIFSTOPPED(status))//if the child is stopped by a signal
```

```
535.             {
536.                 sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
537.                 struct job_t *job = getjobpid(jobs, pidfg);
538.                 job->state = ST;//set the job state
539.                 waiting = false;//stop waiting for fg job
540.                 printf("Job [%d] (%d) stopped by signal %d\n",job->jid,job->pid,WSTOPSIG(
    status));
541.                 fflush(stdout);
542.                 sigprocmask(SIG_SETMASK, &prev_all, NULL);
543.             }
544.
545.             else if(WIFSIGNALED(status))//if the child is terminated by a signal
546.             {
547.                 sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
548.                 struct job_t *job = getjobpid(jobs, pidfg);
549.                 waiting = false;//stop waiting for fg job
550.                 printf("Job [%d] (%d) terminated by signal %d\n",job->jid,job->pid,WTERMS
    IG(status));
551.                 fflush(stdout);
552.                 deletejob(jobs, pidfg);//should deletejob AFTER printing it
553.                 sigprocmask(SIG_SETMASK, &prev_all, NULL);
554.             }
555.             else//if the child is terminated otherwise
556.             {
557.                 sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
558.                 struct job_t *job = getjobpid(jobs, pidfg);
559.                 waiting = false;//stop waiting for fg job
560.                 //printf("Job [%d] (%d) terminated abnormally\n",job->jid,job->pid);
561.                 fflush(stdout);
562.                 deletejob(jobs, pidfg);//should deletejob AFTER printing it
563.                 sigprocmask(SIG_SETMASK, &prev_all, NULL);
564.             }
565.
566.             //In case that the user type a command when running a foreground job
567.             //Clean the stdin buffer after a foreground job's done
568.             //WARNING: This function is optional, you should use it only in one situation
    :
569.             //You would type commands using your human hands, and you want the shell
570.             //ignore your miss type when running a foreground job.
571.             /*
572.             int c;
573.             while ((c=getchar()) != '\n' && c != EOF);
574.             */
575.         }
```

```
576.
577.     }
578.
579.     while((pid = waitpid(-1, NULL, WNOHANG)) > 0)//reap the zombie child
580.     {
581.         sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
582.         if(pid != pidfg)
583.             deletejob(jobs, pid);
584.         sigprocmask(SIG_SETMASK, &prev_all, NULL);
585.     }
586.     /* SIGTSTP default handler can send SIGCHILD, too. Can't check the error here.
587.     if (errno != ECHILD)
588.         perror("waitpid error:");
589.     */
590.     return;
591. }
592.
593. /*
594.  * sigint_handler - The kernel sends a SIGINT to the shell whenver the
595.  *    user types ctrl-c at the keyboard.  Catch it and send it along
596.  *    to the foreground job.
597.  */
598. void sigint_handler(int sig)
599. {
600.     pid_t pid;
601.
602.     if((pid = fgpid(jobs))==0)//if there is no fg job
603.         return;
604.
605.     struct job_t *job = getjobpid(jobs, pid);
606.     kill(-pid, SIGINT);//send the SIGINT to the progress group
607.     if(errno==ESRCH)//error handling
608.     {
609.         perror("kill failed");
610.         exit(1);
611.     }
612.
613.     return;
614. }
615.
616. /*
617.  * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
618.  *     the user types ctrl-z at the keyboard. Catch it and suspend the
619.  *     foreground job by sending it a SIGTSTP.
```

```
 */
void sigtstp_handler(int sig)
{
    pid_t pid;

    if((pid = fgpid(jobs))==0)//if there is no fg job
        return;
    kill(-pid, SIGTSTP);//send the SIGTSP to the progress group
    if(errno==ESRCH)
    {
        perror("kill failed");
        exit(1);
    }

    return;
}

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

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

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

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

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

/* maxjid - Returns largest allocated job ID */
int maxjid(struct job_t *jobs)
{
```

```
664.     int i, max=0;
665.
666.     for (i = 0; i < MAXJOBS; i++)
667.     if (jobs[i].jid > max)
668.         max = jobs[i].jid;
669.     return max;
670. }
671.
672. /* addjob - Add a job to the job list */
673. int addjob(struct job_t *jobs, pid_t pid, int state, char *cmdline)
674. {
675.     int i;
676.
677.     if (pid < 1)
678.     return 0;
679.
680.     for (i = 0; i < MAXJOBS; i++) {
681.     if (jobs[i].pid == 0) {
682.         jobs[i].pid = pid;
683.         jobs[i].state = state;
684.         jobs[i].jid = nextjid++;
685.         if (nextjid > MAXJOBS)
686.         nextjid = 1;
687.         strcpy(jobs[i].cmdline, cmdline);
688.         if(verbose){
689.             printf("Added job [%d] %d %s\n", jobs[i].jid, jobs[i].pid, jobs[i].cmdline);

690.         }
691.         return 1;
692.     }
693.     }
694.     printf("Tried to create too many jobs\n");
695.     return 0;
696. }
697.
698. /* deletejob - Delete a job whose PID=pid from the job list */
699. int deletejob(struct job_t *jobs, pid_t pid)
700. {
701.     int i;
702.
703.     if (pid < 1)
704.     return 0;
705.
706.     for (i = 0; i < MAXJOBS; i++) {
```

```
707.     if (jobs[i].pid == pid) {
708.         clearjob(&jobs[i]);
709.         nextjid = maxjid(jobs)+1;
710.         return 1;
711.     }
712.     }
713.     return 0;
714. }
715.
716. /* fgpid - Return PID of current foreground job, 0 if no such job */
717. pid_t fgpid(struct job_t *jobs) {
718.     int i;
719.
720.     for (i = 0; i < MAXJOBS; i++)
721.     if (jobs[i].state == FG)
722.         return jobs[i].pid;
723.     return 0;
724. }
725.
726. /* getjobpid  - Find a job (by PID) on the job list */
727. struct job_t *getjobpid(struct job_t *jobs, pid_t pid) {
728.     int i;
729.
730.     if (pid < 1)
731.     return NULL;
732.     for (i = 0; i < MAXJOBS; i++)
733.     if (jobs[i].pid == pid)
734.         return &jobs[i];
735.     return NULL;
736. }
737.
738. /* getjobjid  - Find a job (by JID) on the job list */
739. struct job_t *getjobjid(struct job_t *jobs, int jid)
740. {
741.     int i;
742.
743.     if (jid < 1)
744.     return NULL;
745.     for (i = 0; i < MAXJOBS; i++)
746.     if (jobs[i].jid == jid)
747.         return &jobs[i];
748.     return NULL;
749. }
750.
```

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

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

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

    for (i = 0; i < MAXJOBS; i++) {
    if (jobs[i].pid != 0) {
        printf("[%d] (%d) ", jobs[i].jid, jobs[i].pid);
        switch (jobs[i].state) {
        case BG:
            printf("Running ");
            break;
        case FG:
            printf("Foreground ");
            break;
        case ST:
            printf("Stopped ");
            break;
        default:
            printf("listjobs: Internal error: job[%d].state=%d ",
               i, jobs[i].state);
        }
        printf("%s", jobs[i].cmdline);
    }
    }
}
/******************************
 * end job list helper routines
 ******************************/

```

```
795.
796. /***********************
797.  * Other helper routines
798.  ***********************/
799.
800. /*
801.  * usage - print a help message
802.  */
803. void usage(void)
804. {
805.     printf("Usage: shell [-hvp]\n");
806.     printf("   -h   print this message\n");
807.     printf("   -v   print additional diagnostic information\n");
808.     printf("   -p   do not emit a command prompt\n");
809.     exit(1);
810. }
811.
812. /*
813.  * unix_error - unix-style error routine
814.  */
815. void unix_error(char *msg)
816. {
817.     fprintf(stdout, "%s: %s\n", msg, strerror(errno));
818.     exit(1);
819. }
820.
821. /*
822.  * app_error - application-style error routine
823.  */
824. void app_error(char *msg)
825. {
826.     fprintf(stdout, "%s\n", msg);
827.     exit(1);
828. }
829.
830. /*
831.  * Signal - wrapper for the sigaction function
832.  */
833. handler_t *Signal(int signum, handler_t *handler)
834. {
835.     struct sigaction action, old_action;
836.
837.     action.sa_handler = handler;
838.     sigemptyset(&action.sa_mask); /* block sigs of type being handled */
```

```
839.     action.sa_flags = SA_RESTART; /* restart syscalls if possible */
840. 
841.     if (sigaction(signum, &action, &old_action) < 0)
842.     unix_error("Signal error");
843.     return (old_action.sa_handler);
844. }
845. 
846. /*
847.  * sigquit_handler - The driver program can gracefully terminate the
848.  *    child shell by sending it a SIGQUIT signal.
849.  */
850. void sigquit_handler(int sig)
851. {
852.     printf("Terminating after receipt of SIGQUIT signal\n");
853.     exit(1);
854. }
855. 
856. //parse a string to positive integer, if fail then return -1
857. int str2int(char *str)
858. {
859.     int iter = 0;
860.     int res = 0;
861.     if(str[iter]=='\0')
862.         return -1;
863.     while(str[iter]!='\0')
864.     {
865.         if(str[iter]>'9' || str[iter]<'0')
866.             return -1;
867.         res = 10*res+str[iter]-'0';
868.         iter++;
869.     }
870.     return res;
871. }
```