CS 213, Fall 2002
Cs213,2002 年秋季
Lab Assignment L5: Writing Your Own Unix Shell
实验任务 L5: 编写自己的 Unix Shell

Assigned: Oct. 24, Due: Thu., Oct. 31, 11:59PM
分配时间: 10 月 24 日，预定时间: 10 月 31 日，星期四，11:59 PM

Harry Bovik (bovik@cs.cmu.edu) is the lead person for this assignment.
Harry Bovik (Bovik@cs.cmu.edu)是这项任务的负责人。

Introduction
引言

The purpose of this assignment is to become more familiar with the concepts of process control and signalling. You'll do this by writing a simple Unix shell program that supports job control.
这个作业的目的是为了更加熟悉过程控制和信号的概念。你可以通过编写一个简单的支持作业控制的 Unix shell 程序来做到这一点。

Logistics
后勤

You may work in a group of up to two people in solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to t he assignment will be posted on the course Web page.
你可以组成一个最多两人的小组来解决这个任务的问题。唯一的"上交"将是电子的。任何对作业的澄清和修改都会在课程网页上公布。

Hand Out Instructions
发出指示

SITE-SPECIFIC: Insert a paragraph here that explains how the instructor will hand out the `shlab-handout.tar` file to the students. Here are the directions we use at CMU.
SITE-SPECIFIC: 在这里插入一段，解释讲师将如何向学生分发 shlab-handout.tar 文件。以下是我们在卡内基梅隆大学使用的说明。

Start by copying the file `shlab-handout.tar` to the protected directory (the *lab directory*) in which you plan to do your work. Then do the following:
首先，将文件 shlab-handout.tar 复制到您计划在其中执行工作的受保护目录(lab 目录)。然后执行以下操作:

- Type the command `tar xvf shlab-handout.tar` to expand the tarfile.
  输入命令 tar xvf shlab-handout.tar 来展开 tarfile。
- Type the command `make` to compile and link some test routines.
  键入 make 命令以编译和链接一些测试例程。
- Type your team member names and Andrew IDs in the header comment at the top of `tsh.c`.
  在 tsh.c 顶部的头注释中输入你的团队成员名和 Andrew id。

Looking at the `tsh.c` (*tiny shell*) file, you will see that it contains a functional skeleton of a simple Unix shell. To help you get started, we have already implemented the less interesting functions. Your assignment
查看 tsh.c (小 shell)文件，您将看到它包含一个简单 Unix shell 的函数框架。为了帮助你入门，我们已经实现了一些不那么有趣的函数。你的任务

is to complete the remaining empty functions listed below. As a sanity check for you, we've listed the approximate number of lines of code for each of these functions in our reference solution (which includes lots of comments).
就是完成下面列出的剩余的空功能。为了检查您的健全性，我们在参考解决方案中列出了每个函数的大致代码行数(其中包括大量注释)。

- `eval`: Main routine that parses and interprets the command line. [70 lines]
  `Eval`：解析和解释命令行的 `Main` 例程`[70` 行`]`

- `builtin cmd`: Recognizes and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs`. [25 lines]
  `Builtin cmd`：识别和解释内置命令：`quit`，`fg`，`bg`，`and jobs`

- `do bgfg`: Implements the `bg` and `fg` built-in commands. [50 lines]
  `Do bgfg`：实现 `bg` 和 `fg` 的内置命令

- `waitfg`: Waits for a foreground job to complete. [20 lines]
  `Waitfg`：等待前台作业完成。`[20` 行`]`

- `sigchld handler`: Catches SIGCHILD signals. 80 lines]
  捕捉 `SIGCHILD` 信号。`80` 行`]`

- `sigint handler`: Catches SIGINT (`ctrl-c`) signals. [15 lines]
  `SIGINT handler`：`catch SIGINT (ctrl-c) signals`

- `sigtstp handler`: Catches SIGTSTP (`ctrl-z`) signals. [15 lines]
  捕捉 `SIGTSTP (ctrl-z)`信号。`[15` 行`]`

Each time you modify your `tsh.c` file, type `make` to recompile it. To run your shell, type `tsh` to the command line:
每次修改 tsh.c 文件时，输入 make 重新编译它。要运行 shell，在命令行中输入 tsh:

```
unix> ./tsh
Unix > ./tsh
tsh> [type commands to your shell here]
Tsh > [在这里向 shell 键入命令]
```

General Overview of Unix Shells
Unix shell 的一般概述

A *shell* is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeat-edly prints a prompt, waits for a *command line* on `stdin`, and then carries out some action, as directed by the contents of the command line.
Shell 是一个代表用户运行程序的交互式命令行解释器。Shell 重复打印提示符，等待 stdin 上的命令行，然后按照命令行内容的指示执行一些操作。

The command line is a sequence of ASCII text words delimited by whitespace. The first word in the command line is either the name of a built-in command or the pathname of an executable file. The remaining words are command-line arguments. If the first word is a built -in command, the shell immediately executes the

command in the current process. Otherwise, the word is assumed to be the pathname of an executable program. In this case, the shell forks a child process, then loads and runs the program in the context of the child. The child processes created as a result of interpreting a single command line are known collectively as a *job*. In general, a job can consist of multiple child processes connected by Unix pipes.
命令行是由空格分隔的 ASCII 文本单词序列。命令行中的第一个字是一个内置命令的名称或者一个可执行文件的路径名。剩下的单词是命令行参数。如果第一个单词是一个内置命令，shell 会立即在当前进程中执行该命令。否则，这个单词被认为是一个可执行程序的路径名。在这种情况下，shell 分叉一个子进程，然后在子进程的上下文中加载并运行程序。作为解释单个命令行的结果而创建的子进程统称为作业。一般来说，一个作业可以由多个通过 Unix 管道连接的子进程组成。

If the command line ends with an ampersand ” &”, then the job runs in the *background*, which means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line. Otherwise, the job runs in the *foreground*, which means that the shell waits for the job to terminate before awaiting the next command line. Thus, at any point in time, at most one job can be running in the foreground. However, an arbitrary number of jobs can run in the background.
如果命令行以 & 结尾，则作业在后台运行，这意味着 shell 不会等待作业终止后再打印提示符，并等待下一个命令行。否则，作业在前台运行，这意味着 shell 在等待下一个命令行之前等待作业终止。因此，在任何时候，最多只有一个作业可以在前台运行。然而，任意数量的作业可以在后台运行。

For example, typing the command line
例如，键入命令行

```
tsh> jobs
Tsh > jobs
```

causes the shell to execute the built-in `jobs` command. Typing the command line
使 shell 执行内置的 jobs 命令

```
tsh> /bin/ls -l -d
Tsh >/bin/ls-l-d
```

runs the `ls` program in the foreground. By convention, the shell ensures that when the program begins executing its main routine
在前台运行 ls 程序。按照惯例，shell 确保程序开始执行它的主程序时

```
int main(int argc, char *argv[])
Int main (int argc, char * argv [])
```

the `argc` and `argv` arguments have the following values:
Argc 和 argv 参数的值如下:

- `argc == 3`,
  `Argc = = 3`,
- `argv[0] == ‘‘/bin/ls’’`,
  `Argv [0] = =”/bin/ls”`,
- `argv[1]== ‘‘-l’’`,
  `argv [1] = = “-l’`,
- `argv[2]== ‘‘-d’’`.

Alternatively, typing the command line

`Argv [2] = = “-d’”`。或者，

键入命令行

```
tsh> /bin/ls -l -d &
Tsh >/bin/ls-l-d &
```

runs the `ls` program in the background.
在后台运行 ls 程序。

Unix shells support the notion of *job control*, which allows users to move jobs back and forth between background and foreground, and to change the process state (running, stopped, or terminated) of the processes in a job. Typing `ctrl-c` causes a SIGINT signal to be delivered to each process in the foreground job. The default action for SIGINT is to terminate the process. Similarly, typing `ctrl-z` causes a SIGTSTP signal to be delivered to each process in the foreground job. The default action for SIGTSTP is to place a process in the stopped state, where it remains until it is awakened by the receipt of a SIGCONT signal. Unix shells also provide various built-in commands that support job control. For example:
Unix shell 支持作业控制的概念，它允许用户在后台和前台之间来回移动作业，并改变作业中进程的进程状态(运行、停止或终止)。键入 ctrl-c 可以向前台作业中的每个进程发送一个 SIGINT 信号。SIGINT 的默认操作是终止进程。类似地，键入 ctrl-z 会导致一个 SIGTSTP 信号传递给前台作业中的每个进程。

SIGTSTP 的默认操作是将一个进程置于停止状态，直到它被 SIGCONT 信号唤醒。Unix shell 还提供了各种支持作业控制的内置命令。例如:

- `jobs`: List the running and stopped background jobs.
作业： 列出正在运行和已停止的后台作业。

- `bg <job>`: Change a stopped background job to a running background job.
`Bg < job >` ： 将停止的后台作业改为正在运行的后台作业。

- `fg <job>`: Change a stopped or running background job to a running in the foreground.
`Fg < job >` ： 将已停止或正在运行的后台作业更改为在前台运行的作业。

- `kill <job>`: Terminate a job.
杀死 < 作业 > ： 终止作业。

The `tsh` Specification
Tsh 规范

Your `tsh` shell should have the following features:
您的 tsh shell 应该具有以下特性:

- The prompt should be the string “ `tsh>` ”.
•提示符应该是字符串“ `tsh >`”。

- The command line typed by the user should consist of a `name` and zero or more arguments, all sepa-rated by one or more spaces. If `name` is a built-in command, then `tsh` should handle it immediately and wait for the next command line. Otherwise, `tsh` should assume that `name` is the path of an executable file, which it loads and runs in the context of an in itial child process (In this context, the term *job* refers to this initial child process).

用户输入的命令行应该包含一个名称和零个或多个参数，所有参数都由一个或多个空格分隔。如果 name 是一个内置命令，那么 tsh 应该立即处理它并等待下一个命令行。否则，tsh 应该假定 name 是一个可执行文件的路径，它在一个初始子进程的上下文中加载并运行该文件(在这个上下文中，术语 job 指的是这个初始子进程)。

- `tsh` need not support pipes (`|`) or I/O redirection (`<` and `>`).

`Tsh` 不需要支持管道(`|`)或 `i/o` 重定向(`<` 和 `>`)。

- Typing `ctrl-c` (`ctrl-z`) should cause a SIGINT (SIGTSTP) signal to be sent to the current fore-ground job, as well as any descendents of that job (e.g., any child processes that it forked). If there is no foreground job, then the signal should have no effect.

键入 ctrl-c (ctrl-z)应该导致将 SIGINT (SIGTSTP)信号发送到当前前台作业，以及该作业的任何后代(例如，它分叉的任何子进程)。如果没有前台作业，那么该信号应该没有作用。

- If the command line ends with an ampersand `&`, then `tsh` should run the job in the background. Otherwise, it should run the job in the foreground.

如果命令行以 & 结尾，那么 tsh 应该在后台运行作业。否则，它应该在前台运行作业。

- Each job can be identified by either a process ID (PID) or a job I D (JID), which is a positive integer assigned by `tsh`. JIDs should be denoted on the command line by the prefix '`%`'. For example, "`%5`" denotes JID 5, and "`5`" denotes PID 5. (We have provided you with all of the routines you need for manipulating the job list.)

每个作业可以由进程 ID (PID)或作业 ID (JID)标识，后者是由 tsh 分配的一个正整数。Jid 应该在命令行中用前缀"%"表示。例如，"% 5"表示 JID 5，"5"表示 PID 5。(我们已经为您提供了操纵工作列表所需的所有例程

- `tsh` should support the following built-in commands:

`Tsh` 应该支持以下内置命令：

  – The `quit` command terminates the shell.

  Quit 命令终止 shell。

  – The `jobs` command lists all background jobs.

  Jobs 命令列出了所有的后台工作。

  – The `bg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.

  Bg < job > 命令通过发送 SIGCONT 信号重启 < job >，然后在后台运行。参数 < job > 可以是 PID 或 JID。

  – The `fg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground. The `<job>` argument can be either a PID or a JID.

  - fg < job > 命令通过发送 SIGCONT 信号重新启动 < job >，然后在前台运行它。参数 < job > 可以是 PID 或 JID。

- `tsh` should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then `tsh` should recognize this event and print a message with the job's PID and a description of the offending signal.

`Tsh` 应该收获所有的僵尸子代。如果任何作业因为接收到未捕获的信号而终止，那么 `tsh` 应该识别该事件，并打印一条带有作业 `PID` 和违规信号描述的消息。

Checking Your Work
检查你的工作

We have provided some tools to help you check your work.
我们提供了一些工具来帮助你检查你的工作。

Reference solution. The Linux executable `tshref` is the reference solution for the shell. Run this program to resolve any questions you have about how your shell should behave. *Your shell should emit output that is identical to the reference solution* (except for PIDs, of course, which change from run to run).
参考溶液。Linux 可执行文件 tshref 是 shell 的参考解决方案。运行这个程序来解决所有关于 shell 应该如何运行的问题。你的 shell 应该发出与参考解决方案相同的输出(当然，除了 pid，它在运行时会发生变化)。

Shell driver. The `sdriver.pl` program executes a shell as a child process, sends it commands and signals as directed by a *trace file* , and captures and displays the output from the shell.
Shell 驱动程序。Pl 程序作为子进程执行 shell，按照跟踪文件的指示发送命令和信号，并捕获和显示 shell 的输出。

Use the -h argument to find out the usage of `sdriver.pl`:
使用 -h 参数查找 sdriver.pl 的用法:

```
unix> ./sdriver.pl -h
/sdriver.pl-h
Usage: sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
用法: sdriver.pl [-hv ]-t < trace >-s < shellprog >-a < args >
Options:
选项:
  -h            Print this message
  打印这条信息
  -v            Be more verbose
  V 更详细
  -t <trace>    Trace file
  - t < Trace > Trace 文件
  -s <shell>    Shell program to test
  - s < Shell > Shell 程序来测试
  -a <args>     Shell arguments
  - a < args > Shell 参数
  -g            Generate output for autograder
  为 autograder 生成输出
```

We have also provided 16 trace files (`trace{01-16}.txt`) that you will use in conjunction with the shell driver to test the correctness of your shell. The lower-numbered trace files do very simple tests, and the higher-numbered tests do more complicated tests.
我们还提供了 16 个跟踪文件(跟踪{01-16})。Txt)，你将与 shell 驱动程序一起使用来测试你的 shell 的正确性。编号较低的跟踪文件执行非常简单的测试，编号较高的测试执行更复杂的测试。

You can run the shell driver on your shell using trace file `trace01.txt` (for instance) by typing:
你可以使用跟踪文件 trace01.txt (例如)在 shell 上运行 shell 驱动程序，输入以下命令:

```
unix> ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
Unix > ./sdriver.pl-t trace01.txt-s./tsh-a"-p"
```

(the `-a "-p"` argument tells your shell not to emit a prompt), or
(- a"-p"参数告诉你的 shell 不要发出提示)，或者

```
unix> make test01
Unix > make test01
```

Similarly, to compare your result with the reference shell, you can run the trace driver on the reference shell by typing:
同样的，为了比较你的结果和引用 shell，你可以在引用 shell 上运行跟踪驱动:

```
unix> ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
Unix > ./sdriver.pl-t trace01.txt-s./tshref-a"-p"
```

or
或者

```
unix> make rtest01
Unix > make rtest01
```

For your reference, `tshref.out` gives the output of the reference solution on all races. This might be more convenient for you than manually running the shell driver on all trace files.
作为参考，tshref.out 给出了所有种族的参考解决方案的输出。这可能比在所有跟踪文件上手动运行 shell 驱动程序更方便。

The neat thing about the trace files is that they generate the s ame output you would have gotten had you run your shell interactively (except for an initial comment that identifies the trace). For example:
跟踪文件的妙处在于，它们生成的输出与交互式运行 shell 时得到的输出相同(除了标识跟踪的初始注释)。例如:

```
bass> make test15
Bass > make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p"
./sdriver.pl-t trace15.txt-s./tsh-a"-p"
#
# trace15.txt - Putting it all together
Trace15.txt-把它们放在一起
#
tsh> ./bogus
伪造的
./bogus: Command not found.
tsh> ./myspin 10
伪造: Command not found.
tsh > 。/myspin 10
Job (9721) terminated by signal
2 tsh> ./myspin 3 &
Job (9721)由信号 2 tsh > ./myspin
3 & 终止
[1] (9723) ./myspin 3
& tsh> ./myspin 4 &
[1](9723) ./myspin 3 &
tsh > ./myspin 4 &
```

```
[2] (9725) ./myspin 4 &
tsh> jobs
(9725) ./myspin 4 &
tsh > jobs
[1]  (9723) Running  ./myspin 3
[1]  (9723) 跑步  我的旋转 3 &
[2]  (9725) Running  ./myspin 4
[2]  (9725) 跑步  我的旋转 4 &
tsh> fg %1
Tsh > fg% 1
Job [1] (9723) stopped by signal
Job [1](9723)被信号停止 20
tsh> jobs
Tsh > jobs
[1]  (9723) Stopped  ./myspin 3
[1]  (9723) 停了  我的旋转 3 &
[2]  (9725) Running  ./myspin 4
[2]  (9725) 跑步  我的旋转 4 &
tsh> bg %3
Tsh > bg% 3
%3: No such job
% 3: 没有这样的工作
tsh> bg %1
Tsh > bg% 1
[1] (9723) ./myspin 3 &
tsh> jobs
(9723)。/myspin 3 &
tsh > jobs
  [1]  (9723) Running  ./myspin
  [1]  (9723) 跑步  我的旋转 3 &
  [2]  (9725) Running  ./myspin
  [2]  (9725) 跑步  我的旋转 4 &
tsh> fg %1
Tsh > fg% 1
tsh> quit
Tsh > 退出
bass>
鲈鱼 >
```

Hints
提示

- Read every word of Chapter 8 (Exceptional Control Flow) in your textbook.
阅读教科书中第 8 章(异常控制流程)的每一个字。

- Use the trace files to guide the development of your shell. Sta rting with `trace01.txt`, make sure that your shell produces the *identical* output as the reference shell. Then move on to trace file `trace02.txt`, and so on.

使用跟踪文件来指导 shell 的开发。使用 trace01.txt，确保你的 shell 产生与参考 shell 相同的输出。然后继续跟踪文件 trace02.txt，依此类推。

- The `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` functions will come in very handy. The WUNTRACED and WNOHANG options to `waitpid` will also be useful.

Waitpid、 kill、 fork、 execute、 setpgid 和 sigprocmask 函数非常方便。Waitpid 的 WUNTRACED 和 WNOHANG 选项也很有用。

- When you implement your signal handlers, be sure to send `SIGINT` and `SIGTSTP` signals to the en-tire foreground process group, using " `-pid`" instead of " `pid`" in the argument to the `kill` function. The `sdriver.pl` program tests for this error.

在实现信号处理程序时，一定要将 SIGINT 和 SIGTSTP 信号发送到整个前台进程组，在 kill 函数的参数中使用"-pid"而不是" pid"。Pl 程序测试这个错误。

- One of the tricky parts of the assignment is deciding on the allocation of work between the `waitfg` and `sigchld handler` functions. We recommend the following approach:

赋值的一个棘手部分是决定 waitfg 和 sigchld 处理程序函数之间的工作分配。我们推荐以下方法:

  – In `waitfg`, use a busy loop around the `sleep` function.
  在 waitfg 中，在睡眠函数周围使用一个繁忙的循环。

  – In `sigchld handler`, use exactly one call to `waitpid`.
  在 sigchld 处理程序中，对 waitpid 只使用一个调用。

  While other solutions are possible, such as calling `waitpid` in both `waitfg` and `sigchld handler`, these can be very confusing. It is simpler to do all reaping in the handler.
  虽然其他解决方案也是可行的，比如在 waitfg 和 sigchld 处理程序中调用 waitpid，但是这些方法可能会让人很困惑。在处理程序中执行所有收获操作更简单。

- In `eval`, the parent must use `sigprocmask` to block `SIGCHLD` signals before it forks the child, and then unblock these signals, again using `sigprocmask` after it adds the child to the job list by calling `addjob`. Since children inherit the `blocked` vectors of their parents, the child must be sure to then unblock `SIGCHLD` signals before it execs the new program.

在 eval 中，父级必须使用 sigprocmask 在分叉子级之前阻塞 SIGCHLD 信号，然后在通过调用 addjob 将子级添加到作业列表后再次使用 sigprocmask 解除这些信号的阻塞。由于子代继承了父代的阻塞向量，子代必须确保在执行新程序之前解除 SIGCHLD 信号的阻塞。

The parent needs to block the `SIGCHLD` signals in this way in order to avoid the race condition where the child is reaped by `sigchld handler` (and thus removed from the job list) *before* the parent calls `addjob`.
父级需要以这种方式阻塞 SIGCHLD 信号，以避免在父级调用 addjob 之前出现子级被 SIGCHLD 处理程序获取(从而从作业列表中删除)的竞争条件。

- Programs such as `more`, `less`, `vi`, and `emacs` do strange things with the terminal settings. Don't run these programs from your shell. Stick with simple text-based programs such as `/bin/ls`,

诸如 more，less，vi 和 emacs 这样的程序会对终端设置做一些奇怪的事情。不要在 shell 中运行这些程序。坚持使用简单的基于文本的程序，如/bin/ls,
`/bin/ps`, and `/bin/echo`.
`/bin/ps` 和`/bin/echo`。

- When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell then creates a child process, by default that child will also be a member of the

从标准 Unix shell 运行 shell 时，shell 在前台进程组中运行。如果你的 shell 创建了一个子进程，默认情况下这个子进程也是

foreground process group. Since typing `ctrl-c` sends a SIGINT to every process in the foreground group, typing `ctrl-c` will send a SIGINT to your shell, as well as to every process that your shell created, which obviously isn't correct.
前台进程组。由于键入 ctrl-c 将向前台组中的每个进程发送一个 SIGINT，键入 ctrl-c 将向 shell 以及 shell 创建的每个进程发送一个 SIGINT，这显然是不正确的。

Here is the workaround: After the `fork`, but before the `execve`, the child process should call `setpgid(0, 0)`, which puts the child in a new process group whose group ID is identical to the child's PID. This ensures that there will be only one process, your shell, in the foreground process group. When you type `ctrl-c`, the shell should catch the resulting SIGINT and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).
这里有一个解决方案: 在 fork 之后，但是在执行之前，子进程应该调用 setpgid (0,0) ，这将把子进程放在一个新的进程组中，其组 ID 与子进程的 PID 相同。这样可以确保前台进程组中只有一个进程，也就是你的 shell。当键入 ctrl-c 时，shell 应该捕获结果 SIGINT，然后将其转发到适当的前台作业(或者更准确地说，包含前台作业的进程组)。

Evaluation
评估

Your score will be computed out of a maximum of 90 points based on the following distribution:
你的分数将根据以下分布从最高 90 分中计算出来:

80 Correctness: 16 trace files at 5 points each.
正确性: 16 个跟踪文件，每个 5 分。

10 Style points. We expect you to have good comments (5 pts) and to check the return value of EVERY system call (5 pts).
风格点。我们希望你有好的评论(5 分) ，并检查每个系统调用的返回值(5 分)。

Your solution shell will be tested for correctness on a Linux machine, using the same shell driver and trace files that were included in your lab directory. Your shell sho uld produce **identical** output on these traces as the reference shell, with only two exceptions:
您的解决方案 shell 将在 Linux 机器上进行正确性测试，使用与您的实验室目录相同的 shell 驱动程序和跟踪文件。你的 shell 应该在这些跟踪上产生与参考 shell 相同的输出，只有两个例外:

- The PIDs can (and will) be different.

PIDs 可以(也将会)有所不同。

- The output of the `/bin/ps` commands in `trace11.txt`, `trace12.txt`, and `trace13.txt` will be different from run to run. However, the running states of any `mysplit` processes in the output of the `/bin/ps` command should be identical.

Trace11.txt、 trace12.txt 和 trace13.txt 中/bin/ps 命令的输出会因运行而不同。然而，在/bin/ps 命令的输出中，任何 mysplit 进程的运行状态应该是相同的。

Hand In Instructions
交上指示

SITE-SPECIFIC: Insert a paragraph here that explains how the students should hand in their `tsh.c` files. Here are the directions we use at CMU.
SITE-SPECIFIC: 在这里插入一个段落，解释学生应该如何提交 tsh.c 文件。以下是我们在卡内基梅隆大学使用的说明。

• Make sure you have included your names and Andrew IDs in the header comment of `tsh.c`.
请确保在 tsh.c 的头注释中包含了您的姓名和 Andrew id。

• Create a team name of the form:
创建表单的团队名称:

– " ID " where ID is your Andrew ID, if you are working alone, or
- **「身份证」，其中身份证是你的安德鲁身份证，如果你是独自工作，或**

– " ID_1+ID_2" where ID_1 is the Andrew ID of the first team member and ID_2 is the Andrew ID of the second team member.
- **" ID 1 + ID2"id1 是第一组成员的安德鲁 ID id2 是第二组成员的安德鲁 ID。**

We need you to create your team names in this way so that we can autograde your assignments.
我们需要你们以这种方式创建你们的队名，这样我们就可以自动批改你们的作业。

• To hand in your `tsh.c` file, type:

```
make handin TEAM=teamname
```

要递交你的 tsh.c 文件，输入: make

handin TEAM = teamname

where `teamname` is the team name described above.
其中 teamname 是上面描述的团队名。

• After the handin, if you discover a mistake and want to submit a revised copy,

type `make handin TEAM=teamname VERSION=2`

在 handin 之后，如果你发现了一个错误，想要提交一个修改后的副本，输入

make handin TEAM = teamname VERSION = 2

Keep incrementing the version number with each submission.
随着每次提交，版本号不断递增。

• You should verify your handin by looking in
你应该通过查看来验证你的 handin

```
/afs/cs.cmu.edu/academic/class/15213-f01/L5/handin
/afs/cs.cmu.edu/academic/class/15213-f01/L5/handin
```

You have list and insert permissions in this directory, but no read or write permissions.
你在这个目录中有列表和插入权限，但是没有读写权限。

Good luck!
祝你好运！