Lab Assignment: Writing Your Own Shell

Introduction

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.

Hand Out Instruction

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:

- Type the command `tar xvf shlab-handout.tar` to expand the tarfile.
- Type the command `make` to compile and link some test routines.
- Type your team member names and Andrew IDs in the header comment at the top of `tsh.c`.

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

实验任务：编写自己的Shell

介绍

本作业的目的是更熟悉过程控制和信号的概念。您将通过编写一个支持作业控制的简单Unix shell程序来实现这一点。

发放指令

首先，将文件shlab-handout.tar复制到您计划在其中进行工作的受保护目录（实验室目录）。然后执行以下操作：

- 键入命令tar xvf shlab-handout.tar以展开tar文件。
- 键入命令make以编译和链接一些测试例程。
- 在tsh.c顶部的标题注释中键入您的团队成员姓名和Andrew ID。

查看tsh.c（微型shell）文件，您将看到它包含一个简单Unix shell的功能骨架。为了帮助您入门，我们已经实现了不太有趣的功能。作业

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).

- `eval`: Main routine that parses and interprets the command line. [70 lines]
- `builtin_cmd`: Recognizes and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs`. [25 lines]
- `do_bgfg`: Implements the `bg` and `fg` built-in commands. [50 lines]
- `waitfg`: Waits for a foreground job to complete. [20 lines]
- `sigchld_handler`: Catches SIGCHILD signals. 80 lines]
- `sigint_handler`: Catches SIGINT (`ctrl-c`) signals. [15 lines]
- `sigtstp_handler`: Catches SIGTSTP (`ctrl-z`) signals. [15 lines]

Each time you modify your `tsh.c` file, type `make` to recompile it. To run your shell, type `tsh` to the command line:

```
unix> ./tsh
tsh> [type commands to your shell here]
```

General Overview of Unix Shells

A *shell* is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeatedly 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.

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.

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.

For example, typing the command line

```
tsh> jobs
```

是完成下面列出的剩余空功能。作为对您的健全性检查，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：解析和解释命令行的主例程。[70行]
- 内置cmd：_识别并解释内置命令：quit、fg、bg和jobs。[25行]
- do_bgfg：实现bg和fg内置命令。[50行]
- waitfg：等待前台作业完成。[20行]
- sigchld处理程序：捕获SIGCHILD信号。80行]
- sigint处理程序：捕获sigint（ctrl-c）信号。[15行]
- sigtstp处理程序：捕获sigtstp（ctrl-z）信号。[15行]

每次修改tsh.c文件时，键入make重新编译。要运行shell，请在命令行中键入tsh：

```
unix>。/tsh
tsh>[在此处向shell键入命令]
```

Unix shell概述

shell是一个交互式命令行解释器，代表用户运行程序。shell反复打印提示，等待stdin上的命令行，然后按照命令行内容的指示执行某些操作。

命令行是由空格分隔的ASCII文本单词序列。命令行中的第一个单词是内置命令的名称或可执行文件的路径名。剩下的单词是命令行参数。如果第一个单词是内置命令，则shell会立即在当前进程中执行该命令。否则，该词被假定为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。由于解释单个命令行而创建的子进程统称为作业。一般来说，一个作业可以由通过Unix管道连接的多个子进程组成。

如果命令行以与号“&”结束，则作业将在后台运行，这意味着shell在打印提示并等待下一个命令行之前不会等待作业终止。否则，作业将在前台运行，这意味着shell会在等待下一个命令行之前等待作业终止。因此，在任何时间点，前台最多只能运行一个作业。但是，可以在后台运行任意数量的作业。

例如，键入命令行

```
tsh>职位
```

causes the shell to execute the built-in `jobs` command. Typing the command line

```
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

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

the `argc` and `argv` arguments have the following values:

- `argc == 3`,
- `argv[0] == ''/bin/ls''`,
- `argv[1]== ''-l''`,
- `argv[2]== ''-d''`.

Alternatively, typing the command line

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

runs the `ls` program in the background.

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:

- `jobs`: List the running and stopped background jobs.
- `bg <job>`: Change a stopped background job to a running background job.
- `fg <job>`: Change a stopped or running background job to a running in the foreground.
- `kill <job>`: Terminate a job.

The `tsh` Specification

Your `tsh` shell should have the following features:

- The prompt should be the string "`tsh> `".

使shell执行内置的jobs命令。键入命令行

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

在前台运行ls程序。按照惯例，shell确保程序开始执行其主例程时

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

argc和argv参数具有以下值：

- argc＝＝3，
- argv[0]==“/bin/ls”，
- argv[1]==“-l”，
- argv[2]==“-d”。

或者，键入命令行

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

在后台运行ls程序。

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，并更改作业中进程的进程状态（运行、停止或终止）。键入ctrl-c会导致SIGINT信号被传递到前台作业中的每个进程。SIGINT的默认操作是终止进程。同样，键入ctrl-z会将SIGTSTP信号传递给前台作业中的每个进程。SIGTSTP的默认操作是将进程置于停止状态，在该状态下，它将一直保持到收到SIGCONT信号唤醒为止。Unix shell还提供了各种支持作业控制的内置命令。例如：

- jobs：列出正在运行和已停止的后台作业。
- bg<job>：将已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<job>：终止一个作业。

tsh规范

您的tsh外壳应具有以下特征：

- 提示应该是字符串“tsh>”。

- The command line typed by the user should consist of a `name` and zero or more arguments, all separated 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 initial child process (In this context, the term *job* refers to this initial child process).
- `tsh` need not support pipes (`|`) or I/O redirection (`<` and `>`).
- Typing `ctrl-c` (`ctrl-z`) should cause a SIGINT (SIGTSTP) signal to be sent to the current foreground 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.
- 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.
- Each job can be identified by either a process ID (PID) or a job ID (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.)
- `tsh` should support the following built-in commands:
  - The `quit` command terminates the shell.
  - The `jobs` command lists all background jobs.
  - 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.
  - 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.
- `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.

Checking Your Work

We have provided some tools to help you check your work.

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).

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.

Use the -h argument to find out the usage of `sdriver.pl`:

- 用户键入的命令行应由一个名称和零个或多个参数组成，所有参数之间用一个或多个子空格隔开。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应该假设name是可执行文件的路径，它在初始子进程的上下文中加载和运行该文件（在这种上下文中，术语job指的是这个初始子进程）。

- tsh不需要支持管道（|）或I/O重定向（<和>）。
- 键入ctrl-c（ctrl-z）应导致向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）发送SIGINT（SIGTSTP）信号。如果有
  如果没有前台作业，则信号应该没有效果。
- 如果命令行以&和结束，则tsh应在后台运行作业。否则，它应该在前台运行作业。
- 每个作业都可以通过进程ID（PID）或作业ID（JID）来标识，JID是tsh分配的正整数。JID应在命令行上用前缀“%”表示。例如，“%5”
  表示JID 5，“5”表示PID 5。（我们为您提供了操作作业列表所需的所有例程。）
- tsh应支持以下内置命令：
  - quit命令终止shell。
  - jobs命令列出所有后台作业。
  - bg<job>命令通过发送SIGCONT信号重新启动<job>，然后在后台运行它。<job>参数可以是PID或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。<job>参数可以是PID或JID。
- tsh应该把所有的僵尸孩子都收起来。如果任何作业因收到未捕获的信号而终止，则tsh应识别此事件并打印一条包含作业PID和
  对违规信号的描述。

检查你的工作

我们提供了一些工具来帮助您检查工作。

参考解决方案。Linux可执行文件tshref是shell的参考解决方案。运行此程序以解决您对shell行为的任何疑问。您的shell应该发出与参考解决方案相同的输出（当然，PID除外，PID会随着运行而变化）。

壳牌驱动程序。sdriver.pl程序将shell作为子进程执行，按照跟踪文件的指示向其发送命令和信号，并捕获和显示shell的输出。

使用-h参数找出sdriver.pl的用法：

```
unix> ./sdriver.pl -h
Usage: sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
Options:
  -h            Print this message
  -v            Be more verbose
  -t <trace>    Trace file
  -s <shell>    Shell program to test
  -a <args>     Shell arguments
  -g            Generate output for autograder
```

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.

You can run the shell driver on your shell using trace file `trace01.txt` (for instance) by typing:

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

(the `-a "-p"` argument tells your shell not to emit a prompt), or

```
unix> make test01
```

Similarly, to compare your result with the reference shell, you can run the trace driver on the reference shell by typing:

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

or

```
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.

The neat thing about the trace files is that they generate the same output you would have gotten had you run your shell interactively (except for an initial comment that identifies the trace). For example:

```
bass> 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 (9721) terminated by signal 2
tsh> ./myspin 3 &
[1] (9723) ./myspin 3 &
tsh> ./myspin 4 &
```

```
unix>。/sdriver.pl-h
用法：sdriver.pl[-hv]-t<trace>-s<shellprog>-a<args>选项：
  -h              打印此消息
  v               说得更详细些
  -t<trace>       跟踪文件
  -s<shell>       壳牌测试程序
  -a<args>        Shell参数
  -g              生成签名输出
```

我们还提供了16个跟踪文件（trace{01-16}.txt），您将与shell驱动程序结合使用，以测试shell的正确性。编号较低的跟踪文件执行非常简单的测试
编号越高的测试执行的测试越复杂。

您可以使用跟踪文件trace01.txt（例如）在shell上运行shell驱动程序，方法是键入：

```
unix>。/sdriver.pl-t trace01.txt-s。/tsh-a“-p”（-a“-p”参数告诉
```

您的shell不要发出提示），或者unix>make test01

同样，要将结果与引用shell进行比较，您可以通过键入以下命令在引用shell上运行跟踪驱动程序：

```
unix>。/sdriver.pl-t trace01.txt-s。/tshref-a“-p”
```

或

```
unix>制作rtest01
```

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。

跟踪文件的巧妙之处在于，它们生成的输出与您交互式运行shell时得到的输出相同（除了标识跟踪的初始注释）。例如：

```
低音>制作测试15
./sdriver.pl-t trace15.txt-s。/tsh-a“-p”#
#trace15.txt-将所有内容放在一起#
tsh>。/假的
./fastic：找不到命令。tsh>。/myspin
10
作业（9721）因信号2 tsh>而终止。/myspin 3&
[1] (9723) ./myspin 3&tsh>
。/myspin 4&
```

```
[2]
tsh> jobs
[1]
[2]
tsh> fg %1
Job [1] (9723) stopped by signal 20
tsh> jobs
[1] (9723) Stopped    ./myspin 3 &
[2] (9725) Running    ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1]
tsh> jobs
[1]
[2]
tsh> fg %1
tsh> quit
bass>
```

Hints

- Read every word of Chapter 8 (Exceptional Control Flow) firstly please.
- Use the trace files to guide the development of your shell. Starting 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.
- 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.
- When you implement your signal handlers, be sure to send `SIGINT` and `SIGTSTP` signals to the entire foreground process group, using "`-pid`" instead of "`pid`" in the argument to the `kill` function. The `sdriver.pl` program tests for this error.
- 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:
  - In `waitfg`, use a busy loop around the `sleep` function.
  - In `sigchld_handler`, use exactly one call to `waitpid`.

  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.
- 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.

```
2.
tsh>职位[1]
2.
tsh>fg%1
作业[1]（9723）因信号20tsh>作业而停止
[1] （9723）已停止        ./myspin 3&
[2] （9725）跑步          ./myspin 4&
tsh>bg%3
%3:没有这样的工作
tsh>bg%1
1.
tsh>职位[1]
2.
tsh>fg%1 tsh>
退出低音>
```

提示

- 请先阅读第8章（特殊控制流程）的每一个单词。
- 使用跟踪文件来指导shell的开发。从trace01.txt开始，确保您的shell产生与参考shell相同的输出。然后继续跟踪文件
  trace02.txt等。
- waitpid、kill、fork、execve、setpgid和sigprocmask函数将非常方便。等待的WUNTRACED和WNOHANG选项也很有用。
- 当你实现信号处理程序时，一定要向整个前台进程组发送SIGINT和SIGTSTP信号，在kill函数的参数中使用“-pid”而不是“pid”。
  sdriver.pl程序测试此错误。
- 这项任务的一个棘手部分是决定waitfg之间的工作分配
  以及sigchld处理函数。我们建议采用以下方法：
  - 在waitfg中，在sleep函数周围使用忙碌循环。
  - 在sigchld处理程序中，只使用一次waitpid调用。

  虽然其他解决方案也是可能的，例如在waitfg和sigchld处理程序中都调用waitpid，但这些解决方案可能会非常令人困惑。在处理程序中执行所有收割操作更简单。
- 在eval中，父级必须在分叉子级之前使用sigprocmask来阻止SIGCHLD信号，然后在将子级添加到作业列表后再次使用sigprokmask来解锁这些信号
  调用addjob。由于孩子继承了父母被阻止的向量，所以孩子在执行新程序之前必须确保解除对SIGCHLD信号的阻止。

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`.

- 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`, `/bin/ps`, and `/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 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.

  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).

Good Luck!

父级需要以这种方式阻止SIGCHLD信号，以避免在父级调用addjob之前，子级被SIGCHLD处理程序捕获（并因此从作业列表中删除）的竞争情况。

- more、less、vi和emacs等程序对终端设置执行奇怪的操作。不要从shell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和/bin/echo。
- 当您从标准Unix shell运行shell时，您的shell正在前台进程组中运行。如果你的shell随后创建了一个子进程，默认情况下，该子进程也将是
  前台进程组。由于键入ctrl-c会向前台组中的每个进程发送一个SIGINT，因此键入ctrl-c将向您的shell以及您的shell创建的每个进程都发送SIGINT，这显然是不正确的。

  解决方法如下：在fork之后，但在execve之前，子进程应调用setpgid（0,0），这将子进程放入一个新的进程组中，该进程组的组ID与子进程的PID相同。这确保了前台进程组中只有一个进程，即您的shell。当你键入ctrl-c时，shell应该捕获得到的SIGINT，然后将其转发给相应的前台作业（或者更确切地说，包含前台作业的进程组）。

祝你好运