CS 213，2002 年秋季

实验室作业 L5：编写自己的 Unix Shell 已分配：10 月 24 日，到期：10 月 31 日（周四）11:59 下午

Harry Bovik（bovik@cs.cmu.edu) 是这项任务的牵头人。

导言

本作业的目的是进一步熟悉进程控制和标记的概念。为此，您将编写一个支持作业控制的简单 Unix shell 程序。

物流

您最多可以两人一组解决本作业中的问题。唯一的 "交卷 "方式是电子交卷。对作业的任何说明和修改都将公布在课程网页上。

分发说明

> **站点特定：在此插入一段文字，说明教师将如何向学生分发 `shlab-handout.tar` 文件。以下是我们在 CMU 使用的说明。**

首先，将文件 `shlab-handout.tar` 复制到您计划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 `tar xvf shlab-handout.tar` 命令来展开 tar 文件。
- 键入 `make` 命令，编译并链接一些测试例程。
- 在 `tsh.c` 顶部的标题注释中键入团队成员姓名和安德鲁 ID。

查看 `tsh.c`（*tiny shell*）文件，你会发现它包含了一个简单 Unix shell 的功能骨架。为了帮助你入门，我们已经实现了一些不太有趣的函数。你的任务

就是完成下面列出的剩余空函数。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），以便对您进行合理性检查。

- 评估：解析和解释命令行的主要例程。[70行]
- 内置 `cmd`：识别并解释内置命令：`quit`、`fg`、`bg` 和 `jobs`。[25行]
- `do_bgfg`: 执行 `bg` 和 `fg` 内置命令。[50行]
- `waitfg`： 等待前台任务完成。[20行]
- `sigchld_handler`：捕捉 SIGCHILD 信号。80 行]
- `SIGINT_handler`：捕捉 SIGINT (`ctrl-c`) 信号。[15行]
- `SIGTSTP_handler`：捕捉 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> [在此处键入 shell 命令]
```

Unix Shell 概述

shell 是一种交互式命令行解释器，代表用户运行程序。shell 重复打印提示符，等待 `stdin` 上的*命令行*，然后根据命令行的内容执行某些操作。

命令行是以空格分隔的 ASCII 文本单词序列。命令行中的第一个字要么是内置命令的名称，要么是可执行文件的路径名。其余的单词是命令行参数。如果第一个单词是内置命令，shell 会立即在当前进程执行该命令。否则，该单词会被假定为可执行程序的路径名。在这种情况下，shell 会分叉一个子进程，然后在子进程上下文中加载并运行程序。通过解释命令行而创建的子进程统称为*作业*。一般来说，一个作业可以由多个通过 Unix 管道连接的子进程组成。

如果命令行以"&"结尾，则作业在*后台*运行，这意味着 shell 在打印提示并等待下一行命令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 在等待下一行命令之前会等待作业终止。因此，在任何时间点，最多只能有一个作业在前台运行。但是，后台可以运行任意数量的作业。

例如，输入命令行

```
tsh> 工作
```

会导致 shell 执行内置的作业命令。输入命令行

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

在前台运行 `ls` 程序。按照惯例，shell 会确保在程序开始执行其主例程时

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

`argc` 和 `argv` 参数的值如下：

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

或者，输入命令行

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

在后台运行 `ls` 程序。

Unix shell 支持*作业控制*概念，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状态（运行、停止或终止）。输入 `ctrl-c` 会向前台作业中的每个进程发送 SIGINT 信号。SIGINT 的默认操作是终止进程。同样，键入 `ctrl-z` 会向前台任务中的每个进程发送 SIGTSTP 信号。SIGTSTP 的默认操作是将进程置于停止状态，直到收到 SIGCONT 信号将其唤醒为止。Unix shell 还提供了各种支持作业控制的内置命令。例如

- 工作：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运行。
- `kill <job>`（杀死 `<job>`）：终止任务。

`tsh` 规范

您的 `tsh` shell 应具备以下功能：

- 提示符应为字符串 "`tsh>` "。

- 用户输入的命令行应包括一个名称和零个或多个参数，所有参数之间用一个或多个空格隔开。如果 `name` 是内置命令，则 `tsh` 应立即处理，并下一行命令。否则，`tsh` 应假定 `name` 是一个可执行文件的路径，并在初始子进程的上下文中加载和运行该文件（在此上下文中，术语 *job* 就是指初始子进程）。
- `tsh` 不需要支持管道（`|`）或 I/O 重定向（<和>）。
- 键入 `ctrl-c`（`ctrl-z`）将导致向当前前台作业以及该作业的任何子进程（例如该作业分叉的任何子进程）发送 SIGINT（SIGTSTP）信号。如果没有前台作业，则该信号不会产生任何影响。
- 如果命令行以"`&`"结尾，则 `tsh` 应在后台运行任务。否则，它应在前台运行任务。
- 每个任务可以进程 ID (PID) 或任务 ID (JID) 标识，进程 ID 是由 `tsh` 分配的一个正整数。JID 应在命令行中用前缀"`%`"表示。，"`%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：
  - 退出命令会终止 shell。
  - 作业命令会列出所有后台作业。
  - `bg <job>` 命令通过向 `<job>` 发送 SIGCONT 信号重启 `<job>`，然后在后台运行 `<job>`。`<job>` 参数可以是 PID 或 JID。
  - `fg <job>` 命令通过向 `<job>` 发送 SIGCONT 信号重启 `<job>`，然后在前台运行 `<job>`。`<job>` 参数可以是 PID 或 JID。
- `tsh` 应收割其所有僵尸子代。如果有任务因为接收到一个它没有捕捉到的信号终止，那么 `tsh` 应该识别这一事件，并打印一条包含该任务 PID 和违规信号描述的信息。

检查您的工作

我们提供了一些工具来帮助您检查您的工作。

参考解决方案。Linux 可执行程序 `tshref` 是 shell 的参考解决方案。运行该程序可解决任何有关 shell

运行的问题。*您的 shell 输出应该与参考方案完全相同*（当然，PID 除外，PID 在过程中会发生变化）。

shell 驱动程序。`sdriver.pl` 程序将 shell 作为子进程执行，根据*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 参数可了解 `sdriver.pl` 的使用情况：

```
unix> ./sdriver.pl -h
使用方法： sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args> 选项
  -h              打印此信息
  -v              更冗长
  -t <跟踪        跟踪文件
  -s <shell>      要测试的 shell 程序
  -a <args> 参数  shell 参数
  -g              生成 autograder 的输出
```

我们还提供了 16 个跟踪文件（`trace{01-16}.txt`），您可以结合 shell 驱动程序来测试 shell 的正确性。编号较低的跟踪文件进行非常简单的测试，而编号较高的测试则进行更复杂的测试。

您可以在 shell 上使用跟踪文件 `trace01.txt`（）运行 shell 驱动程序，方法是键入

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

数 `-a "-p"` 告诉 shell 不要发出提示），或 `unix> make test01`

同样，要将结果与参考 shell 进行比较，可以在参考 shell 上运行跟踪驱动程序，键入

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

或

`unix> make rtest01`

作为参考，`tshref.out` 提供了所有竞赛的参考解法输出。这可能比在所有跟踪文件上手动运行 shell 驱动程序更方便。

跟踪文件的妙处在于，它们生成的输出与交互式运行 shell 时相同（除了识别跟踪的初始注释）。例如

```
bass> make test15
./sdriver.pl -t trace15.txt -s /tsh -a "-p" #
# trace15.txt - 汇总起来 #
tsh> ./bogus
./bogus：tsh> ./myspin 10
任务 (9721) 因信号 2 而终止 tsh>
./myspin 3 &
[1] (9723) ./myspin 3 &
```

```
tsh> ./myspin 4 &
```

```
[2] (9725) ./myspin 4 &
tsh> 工作
[1] (9723) 运行          ./myspin 3 &
[2] (9725) 运行          ./myspin 4 &
tsh> fg %1
作业 [1] (9723) 因信号 20 而停止 tsh>
jobs
[1] (9723) 已停止        ./myspin 3 &
[2] (9725) 运行          ./myspin 4 &
tsh> bg %3
%3: 没有此类任务 tsh>
bg %1
[1] (9723) ./myspin 3 &
tsh> 工作
[1] (9723) 跑步          ./myspin 3 &
[2] (9725) 跑步          ./myspin 4 &
tsh> fg %1
tsh> quit
bass>
```

提示

- 阅读课本第 8 章（异常控制流）的每一个字。
- 使用跟踪文件来指导 shell 的开发。从 `trace01.txt` 开始，确保你的 shell 产生*与参考 shell 相同*的输出。然后转到跟踪文件 `trace02.txt`，以此类推。
- `waitpid`、`kill`、`fork`、`execve`、`setpgid` 和 `sigprocmask` 函数将非常有用。`waitpid` 的 WUNTRACED 和 WNOHANG 选项也很有用。
- 在执行信号处理程序时，务必向前台进程组发送 `SIGINT` 和 `SIGTSTP` 信号，`kill` 函数的参数中使用"`-pid`"而不是 "`pid`"。`sdriver.pl` 程序会对该错误进行测试。
- 任务的棘手之处之一是决定如何在候补成员之间分配工作。
  和 `sigchld` 处理程序函数。我们建议采用以下方法：
  - 在 `waitfg` 中，围绕睡眠函数使用繁忙循环。
  - 在 `sigchld`-处理程序中，只调用一次 `waitpid`。

  虽然有其他解决方案，例如在 `waitfg` 和 `sigchld` 处理程序中同时调用 `waitpid`，但这些方案可

能会非常混乱。更简单的做法是在处理程序中完成所有重收。

- 在求值过程中，父进程必须在分叉子进程之前使用 `sigprocmask` 阻止 `SIGCHLD` 信号，然后在调用 `addjob` 将子进程添加到作业列表后，再次使用 `sigprocmask` 解除对这些信号的阻止。由于子程序继承了父程序的阻塞向量，因此在执行新程序之前，子程序必须确保解除对 `SIGCHLD` 信号的阻塞。

父进程需要以这种方式阻止 `SIGCHLD` 信号，以避免出现竞赛条件，即子进程*在*父进程调用 `addjob` *之前*就被 `sigchld` 处理程序捕获（从而从作业列表中删除）。

- `more`、`less`、`vi` 和 `emacs` 等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。坚持使用简单的文本程序，如 `/bin/ls`、`/bin/ps` 和 `/bin/echo`。
- 在标准 Unix 中运行 shell 时，shell 会在前台进程组中运行。如果 shell 创建了子进程，默认情况下子进程也是前台进程的成员。由于键入 `ctrl-c` 会向前台进程组中的每个进程发送 SIGINT，因此键入 `ctrl-c` 将您的 shell 以及您的 shell 创建的每个进程发送 SIGINT，这显然是不正确的。

  解决方法如下：在 `fork` 之后、`execve` 之前，子进程应调用 `setpgid(0,0)`，将子进程放入一个新的进程组，其组 ID 与子进程的 PID 相同。这将确保前台进程组中只有一个进程，即你的 shell。当您键入 `ctrl-c` 时，shell 将捕捉到 SIGINT，然后将其转发给相应的前台任务（或者更准确地说，包含前台任务的进程组）。

评估

您的分数将根据以下分布计算，满分 90 分：

80 正确率：16 个跟踪文件，每个文件 5 个点。

10 个风格分。我们希望您有好的注释（5 分），并检查每个系统调用的返回值（5 分）。

我们将在 Linux 机器上使用与实验室目录中相同的 shell 驱动程序和跟踪文件对您的解决方案 shell 进行正确性测试。您的 shell 在这些跟踪文件中应产生与参考 shell **完全相同的**输出，只有两个例外：

- PID 可以（而且将会）不同。
- 每次运行时，`trace11.txt`、`trace12.txt` 和 `trace13.txt` 中 `/bin/ps` 命令的输出会有所不同。不过，`/bin/ps` 命令输出中任何 `mysplit` 进程的运行状态都应相同。

上交说明

具体地点：在此插入一段文字，说明学生应如何他们的 `tsh.c` 文件。以下是我们在 CMU 使用的说明。

- 确保在 `tsh.c` 的头注释中包含您的姓名和安德鲁 ID。
- 创建表单的队名：
  - "*ID*"，其中 *ID* 是您的安德鲁 ID（如果您单独工作），或
  - *"ID* $_1$+ *ID* $_{2)}$"，其中 *ID* $_1$ 是第一名队员的安德鲁 ID，*ID* $_{2)}$ 是第二名队员的安德鲁 ID。

  我们需要您以这种方式创建您的队名，这样我们才能自动分级您的任务。
- 要提交 `tsh.c` 文件，请键入

  使交接队 `TEAM=teamname`

  其中 `teamname` 是上述团队名称。
- 交稿后，如果您发现错误并希望提交，请键入

  ```
  make handin TEAM=teamname VERSION=2
  ```

  每提交一次，版本号就递增一次。
- 您应该在

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

  您在该目录中拥有列表和插入权限，但没有读取或写入权限。

祝你好运