CSC374, Spring 2010
Lab Assignment 1: Writing Your Own Unix Shell

Contact me (`glancast@cs.depaul.edu`) for questions, clarification, hints, etc. regarding this assignment.

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.

Logistics

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 the assignment will be posted on the course Web page.

Hand Out Instructions

Files for this lab are provided on the Linux machines and on the class web site.

- For the Linux machines, log in using your Campus Connection Id and password (or if different, your CDM lab id and password) to

  `cdmlinux.cdm.depaul.edu`

  Copy the file `shlab-handout.tar` to the directory in which you plan to do your work.

  ```
  cp ~glancast/374class/shlab-handout.tar .
  ```

- The shlab-handout.tar file is also available on the class web site. From any Linux machine, you may copy the the shlab-handout.tar file from the web site using the `wget` utility:

```
wget http://condor.depaul.edu/~glancast/374class/hw/shlab-handout.tar
```

After obtaining the shlab-handout.tar file, 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 Campus connection login id into 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 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. [1 line]
- `sigtstp_handler`: Catches SIGTSTP (`ctrl-z`) signals. [1 line]

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
```

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.

The `tsh` Specification

Your `tsh` shell should have the following features:

- The prompt should be the string "`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 tests to help you check your work. Running the tests is not as automated as we would like, but standard techniques of creating scripts to run your shell don't work, since executing within a script changes the notion of the foreground process at inappropriate times. However, there is some assistance as described below.

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).

Makefile. The `Makefile` automates compiling your shell program and 4 programs used to create 16 tests for your shell. The `Makefile` also provides help in running the tests, although this part is not fully automated as noted above.

The lower-numbered tests do simple checks, and the higher-numbered tests do more complicated testing.

Each test consists of a sequence of 1 or more commands (or interrupts) to enter at your shell prompt. In addition each test description has the expected output or effect of executing those commands.

All 16 tests are described in this way in the file `tests.txt` included in the handout tar file. In addition, each the description of each single test is in its own separate file. E.g., test04 is in the file `mtest04.txt`.

To run a test on your shell, it is more convenient to use the 'make' command. The Makefile has a rule that the `make` program can use to help you run each of the 16 tests. To run test07 (for instance), type **make test07** at the Linux prompt. This will generate the following:

```
$ make test07

test07
/**
 * Start the shell ('tsh' or 'tshref')
 * Then enter these Input commands:
 *
 */


Input:
  myspin 8 &
  myspin 9
  ctl-c
  jobs

Expected output:
  tsh> myspin 8 &
  [1] (6106) myspin 8 &
  tsh> myspin 9
  Job [2] (6107) terminated by signal 2
  tsh> jobs
  [1] (6106) Running myspin 8 &
```

```
$ (Now you type tsh or tshref to start your shell and enter the Input command(s))
```

For your reference, `tests.txt` gives the expected output of the reference solution (tshref) for **all** tests or you can use make as above to see the input and expected output for any one test.

Hints

- Read every word of Chapter 8 (Exceptional Control Flow) in your textbook.
- Use the trace files to guide the development of your shell. Starting with `test01`, make sure that your shell produces the *identical* output as the reference shell. Then move on to trace file `test02`, 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.

  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).

Evaluation

Your score will be computed out of a maximum of 175 points based on the following distribution:

160 Correctness: 16 trace files at 10 points each.

15 Style points. We expect you to have comments (5 pts) and to check the return value of EVERY system call (10 pts).

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 should produce **identical** output on these traces as the reference shell, with only two exceptions:

- The PIDs can (and will) be different.
- The output of the `/bin/ps` commands in `test11`, `test12`, and `test13` 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.

Hand In Instructions

- Make sure you have included your names and campus connection login name in the header comment of `tsh.c`.
- Create a team name of the form:
  - "*ID*" where *ID* is your Campus Connection Id, if you are working alone, or
  - "ID_1+ID_2" where ID_1 is the Campus Connection ID of the first team member and ID_2 is the Campus Connection ID of the second team member.

  We need you to create your team names in this way so that we can autograde your assignments.

  Submitting the Assignment On Linux Servers:
- To handin your `tsh.c` file directly from the Linux server, type:

```
make handin TEAM=teamname
```

in the directory where you extracted shlab-handout.tar, where `teamname` is the team name described above.

Note if the team has only one person you can simply write

```
make handin
```

and the teamname will be set to your login name.

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

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

  Examples:

  Student A. Smith submits his `tsh.c` file:

  ```
  make handin TEAM=asmith
  ```

  A. Smith's tsh.c file is copied to the handin directory as

  ```
  asmith-1-tsh.c
  ```

  If VERSION=2 is specified with the make command,

  ```
  make handin TEAM=asmith VERSION=2
  ```

  the tsh.c file for A. Smith will be copied as

  ```
  asmith-2-tsh.c
  ```

  If A. Smith and B. Riley (with login name **briley**) decide to be a team, the team name should be **asmith+briley** (no spaces):

  ```
  make handin TEAM=asmith+briley
  ```

  and the tsh.c file submitted would be copied to the handin directory with file name:

  ```
  asmith+briley-1-tsh.c
  ```

- You should see a message for a successful handin. Also a comment will be added to your assignment on Course Online that the lab has been submitted. After the lab is graded, the comments will be updated and record your score for the assignment.

Submitting from Other Systems:

If you are not using the Linux Servers, you should submit a copy of your `tsh.c` file (only) **renamed** as xxxx-tsh.c where xxxx is your team name as above. Submit this file to Course Online for the shlab assignment.

Good luck!