2015 WINTER OF CODE Project #1: Unix Shell (Due: 11:55 pm, Sunday, March 1)

Introduction

The purpose of this assignment is to become more familiar with the concepts of process control and signaling. You'll do this by writing a simple Unix shell program that supports job control.

Hand Out Instructions

Copy the file `shlab-handout.tar` to the protected 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 name and ID 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 is to complete the remaining empty functions. 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. [80 lines]
- `builtin_cmd`: Recognizes and interprets the built-in commands `quit`, `fg`, `bg`, and `jobs`. [30 lines]
- `do_bgfg`: Implements the `bg` and `fg` built-in commands. [50 lines]
- `waitfg`: Waits for a foreground job to complete. [25 lines]
- `sigchld_handler`: Catches SIGCHILD signals. [75 lines]
- `sigint_handler`: Catches SIGINT (`ctrl-c`) signals. [20 lines]
- `sigtstp_handler`: Catches SIGTSTP (*ctrl-z*) signals. [20 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]
```
If you want to exit your shell, but have not implemented the appropriate commands, you can type `ctrl-D` to kill your shell.

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 prompting for 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 `ls` 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 foreground job.
- `kill <job>`: Terminate a job.

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 small 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.
- `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* (modulo 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.

```
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"
```
or
```
    unix> make test01
```
(The *-a "-p"* argument tells your shell not to emit a prompt.) And, if you want to run all of the tests on your shell, you can type:
```
    unix> make tests
```
Similarly, 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
```
And you can run all the tests on the reference shell by typing
```
    unix> make rtests
```
For your reference, `tshref.out` gives the output of the reference solution on all 16 traces. This might be more convenient for you than running the shell driver.

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:

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

Hints

- Read every word of Chapter 8 (Exceptional Control Flow) in the textbook.
- 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 `psignal`, `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` functions will come in very handy. The WUNTRACED and WNOHANG options to `waitpid` will also be useful.
- 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 `/bin/ls`, `/bin/ps`, and `/bin/echo`. If you run one of these by accident, you *may* be able to get back to normal by typing `stty reset`. In the worst case, you may need to terminate your terminal session.
- 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 solution shell will be tested for correctness on a Linux machine, using the same shell driver and the 16 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 `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.

We reserve the right to run additional checks if we suspect that your program generates the "right" output for the wrong reason.

Turn In Instructions

This project will be submitted via moodle. Be sure that you have:

- Included your full name and email address in the comment at the top of tsh.c
- Removed any extraneous print statements.
- Included any appropriate commentary on your code in a separate README file or as C comments in tsh.c

DUE DATE

11:55 pm, Sunday, March 1.