System Programming

# Coding Project #3. Writing Your Own Unix Shell

## Introduction

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

## What to do and submit

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]

`waitfg`: Waits for a foreground job to complete. [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]
```

## 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.

## 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 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 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 >).

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.

## 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 torun).

**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`:

```
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 10 trace files (`trace 01-10 .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 &
[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 20
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
bass>
```

## Hints

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.

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`.

## Evaluation

Your score will be computed out of a maximum of 100 points based on the following distribution:

**90** Correctness: 10 trace files at 9 points each.
**10** Report : For each function, write 1 ~ 2 page (maximum 2 pages) report explaining each function in files.

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 one exceptions:

The PIDs can (and will) be different.

## Submit

Please upload your all files including tsh.c and report(.pdf).