Tulane University

CMPS 2300, Spring 2023

The Shell Lab: Writing your own Unix Shell

Assigned: February 23, 2023

1st due date: Thursday, March 2 (in lab)

2nd due date: Monday, March 6 (11:59 PM via gradescope)

CMPS 2300 staff (contact via Canvas QR link)

1 Introduction

The purpose of this assignment is for students to become more familiar with the concepts of *processes*, the `fork()` system call, parent and child processes, signalling, and file input/output redirection. You'll do this by writing a simple Unix shell program that supports job control. In addition to learning about many important systems topics, you will also get much more experience writing code in C.

See section 3 for background on shells, and section 4 for specifics on your tasks.

2 Logistics

As usual, this is an individual project. You will hand in your own work and no one else's! The only exception is you may refer to the CS:APP textbook *with citations*. Keep in mind that the examples in the book are not good enough so you cannot use them as-is, they are only a guideline.

When handing in, you will submit only the file `tsh.c` to Gradescope. See Section 8 for handin details.

Attention!

There are *two* due dates. If you have not made sufficient progress by the first due date, you will lose 10 points which you cannot regain! Section 7 will provide more details on how the score is computed. **Do not ignore the first deadline!**

2.1 Hand Out Instructions

We will provide an initial shell program which you will modify. To begin the process of retrieving the initial files, start by logging in to `systems-lab.cs.tulane.edu` as usual.

Your starting point will be provided in the file `shlab-handout.tar` which you will get from our course server. You can copy the file to your own 2300 directory with the following command:

```
$ cp /home/CMPS2300/shlab/shlab-handout.tar ~/2300
```

Once you have obtained the `shlab-handout.tar` file, you can start by doing these things:

- use the command `tar xvf shlab-handout.tar` to expand the tar file, then change into the new directory with `cd shlab-handout`
- use the command `make` to compile some test programs.
- use your editor to put your name and Tulane email address 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 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]
- `do_redirect`: Implements the input/output redirection feature [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:

```
systems-lab$ ./tsh
tsh> [type commands to your shell here]
```

3 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:

```
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.
- `kill <job>`: Terminate a job. (Note: this is *not* a part of the `tsh` spec; see below.)

4 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).
- Program output can be redirected to a file using the standard form (`> output-filename`), and program input can be redirected from a file using the same form (`< input-filename`). (Pipes (`|`) are *not* required for this assignment.)
- 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.

5 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`:

```
systems-lab$ ./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 20 trace files (`trace{01-20}.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:

```
systems-lab$ ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
```

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

```
systems-lab$ make test01
```

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

```
systems-lab$ ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
```

or

```
systems-lab$ make rtest01
```

Helpful tip from a previous student: To see exactly how your shell output differs from the reference shell for a specific test, use this cool trick [1]:

```
systems-lab$ diff <(make test07) <(make rtest07)
```

This will show you how your shell's output from *test07* different from the reference shell. Hopefully this will help you find bugs.

Finally, you can check all of the tests with:

[1] This is a bash-specific trick known as *process substitution*

```
systems-lab$ ./checktsh.py
```

This will notify you of any test failures. You should run this before handing in! This will give you an idea of what score to expect on the correctness test (see Section 7 for an explanation of the scoring).

6 Hints

- Read Chapter 8 (Exceptional Control Flow) in your textbook *closely*. There are examples of code given in the book, which you may use excerpts from in your program, **only if** you explicitly cite the source in a comment. Reminder: you are **not** to copy code from GitHub or any other internet-based source.
- Additional hints, especially on input/output redirection, will be given in class.
- 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`[2] , `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`.

[2]Note that we are referring to the system call `kill` (manual page accessed with the command `man 2 kill`) not the *bash* built-in function `kill` (manual page accessed with the command `man kill` or `man 1 kill`).

- 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).

- Is your shell crashing with a *Segmentation Fault*? This is often due to a null pointer reference involving strings. Fortunately there is a good tool to find out exactly where the problem lies in your code: `gdb`

  To find the line of C code where the crash occurs, you need to run `tsh` within `gdb`.

  ```
  systems-lab$ gdb ./tsh
  (gdb) run
  ```

  Now `tsh` is running within `gdb`. (Hopefully you remember `gdb` from earlier in the course.)

  Once your shell is running, type in the commands that cause it to crash. When the `Segmentation Fault` happens, `gdb` will give you something like this:

  ```
  Program received signal SIGSEGV, Segmentation fault.
  0x00007ffff7a95a6a in vfprintf() from /lib/x86_64-linux-gnu/libc.so.6
  ```

  To find out how the program hit that problem, use the `backtrace` command:

  ```
  (gdb) backtrace
  #0  0x00007ffff7a95a6a in vfprintf () from /lib/x86_64-linux-gnu/libc.so.6
  #1  0x00007ffff7a9e22a in printf () from /lib/x86_64-linux-gnu/libc.so.6
  #2  0x0000000000400c7f in eval (cmdline=<optimized out>) at tsh.c:176
  #3  0x0000000000400ac6 in main (argc=1, argv=0x7fffffffe7d8) at tsh.c:154
  ```

  Look for the first mention of `tsh.c` in this list. This will tell you at which line of source code the bad reference occurred. (In the case above, it is saying line `176` of `tsh.c` is where the crash happens). Start looking for your bug at this point. (The problem may be shortly before the line where the crash occurs, but this should get you close).

7 Evaluation

Your score will be computed out of a maximum of 40 points based on the following distribution:

0 pts, or -10 pts (only on First due date): You must demonstrate a working `eval()` function! This will show correct use of `fork()` to create child processes. **IMPORTANT:** If you do not demonstrate a working `eval()` by the first due date, you will be *penalized* 10 points! You cannot regain these points later.

40 pts (only on Second due date): Pass correctness tests: 20 trace files at 2 points each. Also, as usual **you must write comments which explain your solution in your own words**. Failure to do so will result in serious penalties.

Your solution shell will be tested for correctness on the Gradescope server, using a similar 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 exception: PIDs will be different each time.

8 Hand In Instructions

- Make sure you have included your name and Tulane email in the top comment of `tsh.c`.
- As stated in the previous section, make sure your final submission contains comments explaining all of the code that you added to `tsh.c`. (It is not necessary to explain the code already provided to you, although doing this may help you to understand it.)
- To hand in your `tsh.c` file, use your web browser to log in to Gradescope. Submit *only* your `tsh.c` file. Gradescope will take care of the rest.
- Be sure to check your score in Gradescope after the autograder is finished. Running all traces can take 1 minute or more, perhaps more if many students are submitting.
- You can hand in as often as you like, and check the resulting score in Gradescope. Please verify that your submission worked, and that the output of `checktsh.py` is what you expected.

Please ask questions in class, on the Canvas discussion board, and at course help hours. We don't want you to be stuck. Good luck!