System Programming CSE25101

Coding Project #4. Writing Simple Shell

0. Overview

This assignment is taken in large part from the CMU course that the textbook originates from. Hence, there are many solutions out there in the web. Even so, you should not make use of these solutions. We emphasize again that **plagiarism will not be tolerated** as clearly stated at the beginning of the semester. Do not even attempt to plagiarize!

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 shell program that supports job control.

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 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.

Shells support the notion of *job control*, which allows users to move jobs back and forth between back- ground 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. Shells also provide various built-in commands that support job control.

1. Skeleton Code Download

For this assignment, you should download sp04_assignment.tar from the BlackBoard and move it to your directory in the Linux server(use uni06~10 server). Then, 1) type "Linux*> tar xvf sp04_assignment.tar*" on the command line. This will create a directory that contains a number of files; 2) type "*make*" on this directory.

Looking at the *tsh.c* (***tiny shell***) file, you will see that it contains a functional skeleton of a simple shell(tiny shell). Your assignment is to complete the empty function listed below.

- *eval*: Main routine that parses and interprets the command line.
- *builtin cmd*: Recognizes and interprets two built-in commands, *quit* and *jobs*.
- *waitfg*: Waits for a foreground job to complete.
- *sigint handler*: Catches SIGINT (ctrl-c) signals.
- *sigtstp handler*: Catches SIGTSTP (ctrl-z) signals.

Each time you modify your *tsh.c* file, type make to recompile it. To run your shell, type tsh to the command line:

```
Linux> ./tsh
tsh> [type commands to your shell here]
```

2. Description

2.1 The *tsh* Specification

- 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).
- Typing *ctrl–c* (*ctrl–z*) should cause a SIGINT (SIGTSTP) signal to be sent to the current foreground job, as well as any descendants 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.

- *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.

2.2 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*

```
Linux> ./sdriver.pl -h
```

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 *trace08.txt* (for instance) by typing:

```
Linux> ./sdriver.pl -t trace08.txt -s ./tsh -a "-p"
```

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

Linux> *make test08*

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

Linux> *./sdriver.pl -t trace08.txt -s ./tshref -a "-p"*

or

Linux> *make rtest08*

For your reference, *tshref.out* gives the output of the reference solution on all traces. 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).

2.3 Hint

- 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 *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.
- 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.

- 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 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).

2.4 Evaluation

Correctness:10 trace files at 10 points each.

Your solution shell will be tested for correctness, using the same shell driver and trace files that were included in your assignment 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 trace08.txt and trace09.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.

3. Submission

Each time you type make, the *Makefile* creates a tar file, called *sp04_0000000.tar*, that contains your current *tsh.c* files. Rename it to "*sp04_your-student-id.tar*" (sp04_20161111.tar) and submit it on Blackboard by 11:59pm on Dec. 21. NO LATE SUBMISSIONs will be accepted. Early

submissions will be accepted starting from 00:00am of the 11h and will receive 1 bonus point per day.

4. Q&A

If you have any questions, write your question on “Discussion Board” on BlackBoard in English so that it can be shared, and then send an email to the TA. Shared meaningful questions will get additional bonus points.