CSE 109 – Systems Software

Programming Lab 2

Due: 2/18/2019

Tiny Shell (tsh)

Background

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.

You will implement the following functions in a skeleton tiny shell application [nominal line counts of fully implemented functions are indicated]:

- The main event loop [20 lines]
- eval: Main routine that parses and interprets the command line. [70 lines]
- builtin cmd: Recognizes and interprets the built-in commands: quit and jobs. [25 lines]
- waitfg: Waits for a foreground job to complete. [20 lines]
- sigint handler: Catches SIGINT (ctrl-c) signals. [15 lines]
- sigtstp handler: Catches SIGTSTP (ctrl-z) signals. [15 lines]

Each time you modify your tsh.c file, type `gcc tsh.c -o tsh` to recompile it. To run your shell, type tsh to the command line:

```
unix> ./tsh

tsh> [type commands to your shell here]
```

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 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.

Examples

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.

Hints:

- 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.
- 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. 6 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).