P1: Writing Your Own Unix Shell

CS 283 Systems Programming

Grading: 100 Points

Introduction

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

Instructions

Copy the file shell-handout.zip to your computer.

The tsh.c (tiny shell) file 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. The list shows in [ ] the approximate number of lines of code for each of these functions including comments.

- eval: Main routine that parses and interprets the command line. [70]
- builtin_cmd: Recognizes and interprets the built-in commands:
- quit, fg, bg, and jobs. [25]
- do_bgfg: Implements the bg and fg built-in commands. [50]
- waitfg: Waits for a foreground job to complete. [20]
- sigchld_handler: Catches SIGCHILD signals. [80]
- sigint_handler: Catches SIGINT (ctrl-c) signals. [15]
- sigtstp_handler: Catches SIGTSTP (ctrl-z) signals. [15]

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

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

In addition, shell-handout.zip contains four small programs that you can use to test various aspects of your shell program:

- myspin.c: Takes argument <n> and spins for <n> seconds
- mysplit.c: Forks a child that spins for <n> seconds
- mystop.c: Spins for <n> seconds and sends SIGTSTP to itself
- myint.c: Spins for <n> seconds and sends SIGINT to itself

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 the argc and argv arguments have the following values:

```
int main(int argc, char *argv[])
```

- `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.

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 shall support pipes (|) and I/O redirection (< and >). Only a single pipe must be tested (a < foo | b > bar).
- 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.

Checking 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 produce output that is identical to the reference solution (except for PIDs, of course, which change from run to run).

Hints

- Read every word of Chapter 8 (Exceptional Control Flow) in your textbook.
- 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.
- 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.
- 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).

- Remember that you can use functions like strtok() or makeargv() to parse command lines into string arrays.

Code Organization

Generally speaking, your code will look something like this:

```
while (1)
 {
  read a line of input
  cmd = parse command line
  pid = fork();
  if (pid == 0)
   {
    extract the program name from cmd
    ...
    exec( ... args ...);
   }
   else
    {
     wait(&status);
     check return code placed in status;
    }
 }
```

Evaluation

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

Correctness 90: Your solution shell will be tested for correctness on tux.

Style points 10: have good comments (5 pts), check the return value of EVERY system call (5 pts).