PKU-ICS
Shell Lab: Writing Your Own Linux Shell

1 Introduction

The purpose of this assignment is to get more familiar with the concepts of process control and signalling. You'll do this by writing a simple Linux shell program that supports a simple form of job control and I/O redirection. Please read the whole writeup before starting.

2 Logistics

This is an individual project. You can do this lab on the ICS Linux machines.

3 Hand Out Instructions

Download the file `tshlab-handout.tar` from Autolab, and copy it to the protected directory (the *lab directory*) in which you plan to do your work. Then do the following *on a linux machine*:

- Type the command `tar xvf tshlab-handout.tar` to expand the tar-file.
- Type your name and Student ID in the header comment at the top of `tsh.c`.
- Type the command `make` to compile and link the driver, the trace interpreter, and the test routines.

Looking at the `tsh.c` (*tiny shell*) file, you will see that it contains a skeleton of a simple Linux shell. It will not, of course, function as a shell if you compile and run it now. To help you get started, we have already implemented the less interesting functions such as the routines that manipulate the job list and the command line parser. 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, this is a good thing).

- `eval`: Main routine that parses and interprets the command line. [400 lines, including some helper functions]
- `sigchld_handler`: Catches SIGCHLD signals. [100 lines]

- `sigint_handler`: Catches SIGINT (`ctrl-c`) signals. [30 lines]
- `sigtstp_handler`: Catches SIGTSTP (`ctrl-z`) signals. [30 lines]

When you wish to test your shell, type `make` to recompile it. To run it, type `tsh` to the command line:

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

4 General Overview of Linux 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 Linux pipes. However, the shell you write in this lab need not support 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.

Linux 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. For example,

- 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.

Linux 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 into a running background job.
- `fg` *job*: Change a stopped or running background job into a running foreground job.
- `kill` *job*: Kill a job in the job list.
- `nohup` *[command]*: Make the trailing command block any SIGHUP signals.

Linux shells also support the notion of **I/O redirection**, which allows users to redirect `stdin` and `stdout` to disk files. For example, typing the command line

```
tsh> /bin/ls > foo
```

redirects the output of `ls` to a file called `foo`. Similarly,

```
tsh> /bin/cat < foo
```

displays the contents of file `foo` on `stdout`.

5 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). If you are running system programs like `ls`, you will need to enter the full path (in this case `/bin/ls`) because your shell does not have search paths.
- `tsh` need not support pipes (`|`), but **MUST** support I/O redirection ("<" and ">"), for instance:

  ```
  tsh> /bin/cat < foo > bar
  ```

  Your shell must support both input and output redirection in the same command line.
- Typing `ctrl-c` (`ctrl-z`) should cause your shell to send a SIGINT (SIGTSTP) signal 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.
  - 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.
  - The `kill` *job* command kills a *job* in the job list, or a process group by sending each relevant process a SIGTERM signal. The *job* argument can be either a PID or a JID. Note that if you get a negative *job* argument, such as `kill %-1` or `kill -15213`, your shell should kill the process group of the job with a JID of *%-JID*, or of the job with a PID of *-PID*. If the process group does not exist, your shell should print "*%JID*: No such process group" or "(*PID*): No such process group", where *JID* and *PID* should be replaced by the command line argument. On the

other hand, if the *job* argument is positive, your shell should kill the corresponding job. If the *job* does not exist, your shell should print "*%JID*: No such job" or "(*PID*): No such process". Play with the reference shell to check the details and gain intuition.

Note: The built-in command `kill` slightly differs from that of the Linux shell in semantics, since our shell always kills a *job* rather than a single process.

- The `nohup` *`[command]`* command makes the trailing command ignore any SIGHUP signals. For simplcity, your shell does not need to support built-in commands following this principle. Instead, you can assume *[command]* to be the path of an executable file followed by its arguments.

- Your shell should be able to redirect the output from the `jobs` built-in command. For example

```
tsh> jobs > foo
```

- `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.

6 Checking Your Work

Running your shell. The best way to check your work is to run your shell from the command line. Your initial testing should be done manually from the command line. Run your shell, type commands to it, and see if you can break it. Use it to run real programs!

Reference solution. The 64-bit Linux executable `tshref` is the reference solution for the shell. Run this program (on a 64-bit machine) 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).

Once you are confident that your shell is working, then you can begin to use some tools that we have provided to help you check your work more thoroughly. (These are the same tools that the autograder will use when you submit your work for credit.)

Trace interpreter. We have provided a set of trace files (`trace*.txt`) that validate the correctness of your shell (the appendix section at the end of this handout describes each trace file briefly). Each trace file tests one shell feature. For example, does your shell recognize a particular built-in command? Does it respond correctly to the user typing a `ctrl-c`?.

The `runtrace` program (the trace interpreter) interprets a set of shell commands specified by a single trace file:

```
linux> ./runtrace -h
Usage: runtrace -f <file> -s <shellprog> [-hV]
Options:
  -h            Print this message
  -s <shell>    Shell program to test (default ./tsh)
  -f <file>     Trace file
  -V            Be more verbose
```

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:

```
linux> ./runtrace -f trace05.txt -s ./tsh
#
# trace05.txt - Run a background job.
#
tsh> ./myspin1 &
[1] (15849) ./myspin1 &
tsh> quit
```

The lower-numbered trace files do very simple tests, and the higher-numbered tests do increasingly more complicated tests.

Shell driver. After you have used `runtrace` to test your shell on each trace file individually, then you are ready to test your shell with the shell driver. The `sdriver` program uses `runtrace` to run your shell on each trace file, compares the output to the output produced by the reference shell, and displays the `diff` if they differ.

```
linux> ./sdriver -h
Usage: sdriver [-hV] [-s <shell> -t <tracenum> -i <iters>]
Options
-h            Print this message.
-i <iters>    Run each trace <iters> times (default 4)
-s <shell>    Name of test shell (default ./tsh)
-t <n>        Run trace <n> only (default all)
-V            Be more verbose.
```

Running the driver without any arguments tests your shell on all of the trace files. To help detect race conditions in your code, the driver runs each trace multiple times. You will need to pass each of the tests to get credit for a particular trace:

```
linux> ./sdriver
Running 4 iters of trace00.txt
1. Running trace00.txt...
2. Running trace00.txt...
3. Running trace00.txt...
4. Running trace00.txt...
Running 4 iters of trace01.txt
1. Running trace01.txt...
2. Running trace01.txt...
3. Running trace01.txt...
4. Running trace01.txt...
Running 4 iters of trace02.txt
1. Running trace02.txt...
2. Running trace02.txt...
3. Running trace02.txt...
4. Running trace02.txt...

...
```

```
Running 4 iters of trace30.txt
1. Running trace30.txt...
2. Running trace30.txt...
3. Running trace30.txt...
4. Running trace30.txt...
Running 4 iters of trace31.txt
1. Running trace31.txt...
2. Running trace31.txt...
3. Running trace31.txt...
4. Running trace31.txt...

Score: 128/128
```

Use the optional `-i` argument to control the number of times the driver runs each trace file:

```
linux> ./sdriver -i 1
Running trace00.txt...
Running trace01.txt...
Running trace02.txt...
Running trace03.txt...

...

Running trace30.txt...
Running trace31.txt...

Score: 128/128
```

Use the optional `-t` argument to test a single trace file:

```
linux> ./sdriver -t 06
Running trace06.txt...
Success: The test and reference outputs for trace06.txt matched!
```

Use the optional `-V` argument to get more information about the test:

```
linux> ./sdriver -t 06 -V
Running trace06.txt...
Success: The test and reference outputs for trace06.txt matched!
Test output:
#
# trace06.txt - Run a foreground job and a background job.
#
tsh> ./myspin1 &
[1] (10276) ./myspin1 &
tsh> ./myspin2 1

Reference output:
```

```
#
# trace06.txt - Run a foreground job and a background job.
#
tsh> ./myspin1 &
[1] (10285) ./myspin1 &
tsh> ./myspin2 1
```

Note: The driver program runs the reference shell, which is a 64-bit binary, and thus will not run on a 32-bit machine.

7 Warnings

- Start early! Leave yourself plenty of time to debug your solution, as subtle problems in your shell are hard to find and fix.
- Be careful about race conditions on the job list. Remember that you cannot make any assumptions about the order of execution of the parent and child after forking. In particular, you cannot assume that the child will still be running when the parent returns from the `fork`. In fact, our driver has code that purposely introduces non-determinism in the order that the parent and child execute after forking. Also, remember that signal handlers run concurrently with the program and can interrupt it anywhere, unless you explicitly block the receipt of the signals.
- Remember that simply passing the tests multiple times does not prove the correctness of your shell. We will deduct correctness points if there are race conditions in your code, so it is in your best interest to find them before we do.
- It is forbidden to spin in a tight loop while waiting for a signal (e.g., "`while (1);`"). Doing so is extremely wasteful of processor time. Calling `sleep` inside a tight loop is not appropriate either. Instead, you should use the `sigsuspend` function inside any tight loops. See the textbook for details.
- When children of your shell die, they must be reaped within a bounded amount of time. This means that you can not wait until the foreground process finishes or a user input is entered before reaping.
- You should not call `waitpid` in multiple places. This sets you up for a ton of potential race conditions and makes your shell needlessly complicated.
- Don't use any system calls (e.g., `tcsetpgrp`) that manipulate terminal groups, which will break the autograder.

8 Hints

- Read and understand every word of Chapter 8 (Exceptional Control Flow) and Chapter 10 (System-level I/O) in your textbook.

- Read the code in `tsh.c` carefully before you start. Understand the high-level control flow, and get familiar with the defined global variables and the helper routines.
- Play with your shell by typing commands to it directly. Don't make the mistake of running the trace generator and driver immediately. Develop some familiarity and intuition about how your shell works before testing it with the automated tools.
- Only after you have tested your shell directly from the command line and are fairly confident that it is correct should you start testing with the `runtrace` and driver programs.
- Use the trace files to guide the development of your shell. Starting with `trace00.txt`, make sure that your shell produces the *identical* output as the reference shell. Then move on to trace file `trace01.txt`, and so on.
- The `waitpid`, `kill`, `fork`, `execve`, `setpgid`, `sigprocmask`, and `sigsuspend` functions will come in very handy. The WUNTRACED and WNOHANG options to `waitpid` will also be useful. Use `man` to check out the details about each function.
- 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 driver program specifically tests for this error.
- One of the tricky parts of the assignment is deciding on the allocation of work between the `eval` and `sigchld_handler` functions when the shell is waiting for a foreground job to finish. For example with `trace31` esepcially, you may need to think carefully when and where to update a job's state.
- In order to avoid deadlock, it is recommended to only invoke async-signal-safe functions in your handler. Please refer to Section 8.5.5 in your textbook for a full understanding. For your convenience, we have provided you with a lightweight safe I/O function `sio_put` which reads a format string just like `printf` of C standard, formats it and prints it to the standard output. It supports two kinds of escaping characters, namely `%d` (to print an `int`) and `%%` (to print `%`). Its implementation lies in `tsh.c` in your handout. You can regard it as a handy and secure substitute for `sio_puts` and `sio_putl`.
- In `eval`, the parent must use `sigprocmask` to block `SIGCHLD`, `SIGINT`, and `SIGTSTP` 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 these signals before it `execs` the new program. The child should also restore the default handlers for the signals that are ignored by the shell.

  The parent needs to block signals in this way in order to avoid race conditions (e.g., the child is reaped by `sigchld_handler` (and thus removed from the job list) *before* the parent calls `addjob`). Section 8.5.6 has details about the race conditions and how to block signals explicitly.
- Programs such as `top`, `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/cat`, `/bin/ls`, `/bin/ps`, and `/bin/echo`.

- When you run your shell from the standard Linux 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). [1]

9 Evaluation

Your score will be computed out of a maximum of 128 points based on the following distribution:

128 Correctness: 32 trace files at 4 pts each. In addition, if your solution passes the traces but is not actually correct (you hacked a way to get it to pass the traces), we will deduct correctness points during our read through of your code.

The most common thing we will be looking for is race conditions that you have simply plastered over, often using the `sleep` call. In general you code should not have races even if we remove all `sleep` calls.

10 Style points. We expect you to follow the style guidelines at

`https://www.cs.cmu.edu/~213/codeStyle.html`

For example, we expect you to have good comments and to check the return value of EVERY system call. We also expect you to break up large functions such as `eval` into smaller helper functions, to enhance readability and avoid duplicating code. Some advice about commenting:

- Do begin your program file with a descriptive block comment that describes your shell.
- Do begin each routine with a block comment describing its role at a high level.
- Do preface related lines of code with a block comment.
- Do keep your lines within 80 characters.
- Don't simply comment each line.

You should also follow other guidelines of good style, such as using a consistent indenting style (don't mix spaces and tabs!), using descriptive variable names, and grouping logically related blocks of code with whitespace.

[1] Note that this is a simplification of the way that real shells work. With real shells, the kernel responds to `ctrl-c` (`ctrl-z`) by sending SIGINT (SIGTSTP) directly to each child process in the terminal foreground process group. The shell manages the membership of this group using the `tcsetpgrp` function, and manages the attributes of the terminal using the `tcsetattr` function, both of which are outside the scope of this class and would break our current autograding scheme.

Your solution shell will be tested for correctness on a 64-bit ubuntu machine (the Autolab server), using the same driver and trace files that were included in your handout 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 `trace19.txt`, `trace20.txt`, and `trace21.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.

The driver deals with all of these subtleties when it checks for correctness.

10 Hand In Instructions

- Make sure you have included your name and Student ID in the header comment of `tsh.c`.
- Hand in your `tsh.c` file for credit by uploading it to Autolab. You may hand in as often as you like. You will be graded on the **last** version you hand in.
- After you hand in, it takes a minute or two for the driver to run through multiple iterations of each trace file.
- We'll be using a sophisticated cheat checker that compares handins from this year and previous years. Please don't copy another student's code. Start early, and if you get stuck, come see your instructors for help.

Good luck!

Appendix: Trace Files

The trace driver runs an instance of your shell in a child process and communicates with the shell interactively in a way that mimics the behavior of a user. To test the behavior of your shell, the trace driver reads in trace files that specify shell line commands (that are actually sent to the shell) as well as a few special synchronization commands (that are interpreted by the driver when handling the shell process). The trace files may also reference a number of shell test programs to perform various functions, and you may refer to the code and comments of these test programs for more information.

The format of the trace files is as follows:

- The comment character is `#`. Everything to the right of it on a line is ignored.
- Each trace file is written so that the output from the shell shows exactly what the user typed. We do this by using the `/bin/echo` program, which not only tests the shell's ability to run programs, but also shows what the user typed. For example:

  ```
  /bin/echo -e tsh\076 ./myspin1 \046
  ```

  Note: octal `\076` is `>` and octal `\046` is `&`. These are special shell metacharacters that need to be escaped. This line represents `tsh> ./myspin1 &`, that is, a user trying to run `./myspin1` in the background.
- There are also a few special commands for synchronization between the job (your shell) and the parent process (the driver) and to send Linux signals from the parent to the job.

WAIT	Wait for a sync signal from the job over its synchronizing domain socket.
SIGNAL	Send a sync signal to the job over its synchronizing domain socket.
3*NEXT	Read and print all responses from the shell until you see the next shell prompt. This command is essential for synchronizing with the shell and mimics the way people wait until they see the shell prompt until they type the next string.
SIGINT	Send a SIGINT signal to the job.
SIGTSTP	Send a SIGTSTP signal to the job.

The following table describes what each trace file tests on your shell against the reference solution.

NOTE: this table is provided so that you can quickly get a high level picture about the testing traces. The explanation here is over-simplified. To understand what exactly each trace file does, you need to read the trace files.

trace00.txt	Properly terminate on EOF.
trace01.txt	Process built-in quit command.
trace02.txt	Run a foreground job that prints an environment variable.
trace03.txt	Run a synchronizing foreground job without any arguments.
trace04.txt	Run a foreground job with arguments.
trace05.txt	Run a background job.
trace06.txt	Run a foreground job and a background job.
trace07.txt	Use the jobs built-in command.
trace08.txt	Send fatal SIGINT to foreground job.
trace09.txt	Send SIGTSTP to foreground job.
trace10.txt	Send fatal SIGTERM (15) to a background job.
trace11.txt	Child sends SIGINT to itself.
trace12.txt	Child sends SIGTSTP to itself.
trace13.txt	Forward SIGINT to foreground job only.
trace14.txt	Forward SIGTSTP to foreground job only.
trace15.txt	Process bg built-in command (one job).
trace16.txt	Process bg built-in command (two jobs).
trace17.txt	Process fg built-in command (one job).
trace18.txt	Process fg built-in command (two jobs).
trace19.txt	Forward SIGINT to every process in foreground process group.
trace20.txt	Forward SIGTSTP to every process in foreground process group.
trace21.txt	Restart every stopped process in process group.
trace22.txt	I/O redirection (input).
trace23.txt	I/O redirection (input and output).
trace24.txt	I/O redirection (input and output, but different order).
trace25.txt	Nohup (send SIGHUP to normal and nohup command).
trace26.txt	Kill (kill a job that is not in job list).
trace27.txt	Kill (kill a job by JID).
trace28.txt	Invoke SIGTERM handler by processing kill built-in command.
trace29.txt	Send SIGTSTP to a foreground job, then switch it to background.
trace30.txt	Kill (kill a process group by JID).
trace31.txt	Robustness (continue a process from outside your shell).