TinyShell

1 Introduction

The purpose of this assignment is to 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 and I/O redirection.

2 Hand Out Instructions

Copy tshlab-handout.tar to the protected directory (the *lab directory*) in which you plan to do your work. Then do the following:

- Type the command tar xvf tshlab-handout.tar to expand the tar-file.
- Type the command make to compile and link the driver, the trace interpreter, and the test routines.
- Type your name(s) and Andrew ID 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 skeleton of a simple Unix shell. 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).

- eval: Main routine that parses and interprets the command line. [300 lines, including some helper functions]
- 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]

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]
```

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 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.

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. 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.

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 into a running background job.
- fg *job*: Change a stopped or running background job into a running foreground job.
- kill *job*: Terminate a job.

Unix 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.

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).
- tsh need not support pipes (|), but **MUST** support I/O redirection ("<" and ">"). For simplicity, assume that "<" and ">" are always surrounded by one or more spaces on the command line. For example, you can always assume input of the form

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

and never

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

- 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.
  - 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.
- 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.

5 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 interpretor) 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, displays the diff if they differ, and optionally sends the results to the Autolab server:

```
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 5)
   -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 5 iters of trace00.txt
1. Running trace00.txt...
2. Running trace00.txt...
3. Running trace00.txt...
4. Running trace00.txt...
5. Running trace00.txt...
Running 5 iters of trace01.txt
1. Running trace01.txt...
2. Running trace01.txt...
3. Running trace01.txt...
4. Running trace01.txt...
5. Running trace01.txt...
Running 5 iters of trace02.txt
1. Running trace02.txt...
2. Running trace02.txt...
3. Running trace02.txt...
4. Running trace02.txt...
5. Running trace02.txt...
...
Running 5 iters of trace23.txt
1. Running trace23.txt...
2. Running trace23.txt...
3. Running trace23.txt...
4. Running trace23.txt...
5. Running trace23.txt...
Running 5 iters of trace24.txt
1. Running trace24.txt...
2. Running trace24.txt...
3. Running trace24.txt...
4. Running trace24.txt...
5. Running trace24.txt...
Summary: 25/25 correct traces
```

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 trace23.txt...
Running trace24.txt...
Summary: 25/25 correct traces
```

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!
```

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

6 Hints

- Read and understand every word of Chapter 8 (Exceptional Control Flow) and Chapter 11 (Systemlevel I/O) in your textbook.
- Read the code in tsh.c carefully before you start. Understand the high-level control flow, 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 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.
- 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.

- 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. 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. We recommend the following approach:
  - In eval (or one of its helpers), use a busy wait loop around the sleep function.
  - In sigchld_handler, there should be exactly one instance of waitpid (in a loop of course to handle the possibility that multiple children need to be reaped).

  While other solutions are possible, such as calling waitpid in both eval 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, 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 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 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/cat, /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 sjob).

Appendix

The following table describes what each trace file tests.

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)