ECE 322 - Systems Programming

University of Massachusetts Amherst
The College of Engineering
Department of Electrical and Computer Engineering

Assignment 3

You may work with a partner on this assignment. Working with a partner is voluntary and simply means that you can submit the same code and it will not count as a violation of the UMass Academic Integrity Policy. If only 1 partner uploads the code, then the code must have the names of both partners at the top. If you used AI in your assignment, you should also include a statement in comments at the top of your code that describes how you used it.

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 Unix shell program that supports job control.

Download the file for the assignment here. As with the previous assignments, all work should be completed on the class VM. To download the assignment tar file from the terminal in the class VM and untar it, run the following commands.

For those using ARM64 Linux on VMware Fusion, replace the executable binary tshref with version here. That is, download this file into the assignment3 directory, and run "cp tshref-arm tshref."

$ wget --user irwin-ece --password myclass --no-check-certificate https://courses.umass.edu/eceng322-deirwin/fall25/assignments/assignment3.tar.gz
$ tar xzvf assignment3.tar.gz

The commands above will create an assignment3/ directory, which has the assignment files. The key file here is tsch.c. Looking at the tsh.c (tiny shell) file, you will see that it 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. 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 below (which includes lots of comments).

- **eval:** Main routine that parses and interprets the command line. [70 lines]

- **builtin_cmd:** Recognizes and interprets the built-in commands: quit, fg, bg, and jobs. [25 lines]

- **do_bgfg:** Implements the bg and fg built-in commands. [50 lines]

- **waitfg:** Waits for a foreground job to complete. [20 lines]

- **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 on the command line:

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

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 the 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 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 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:
  1. The **quit** command terminates the shell.
  2. The **jobs** command lists all background jobs.
  3. The **bg <job>** command restarts <job> by sending it a SIGCONT signal, and then runs it in the background. The <job> argument can either be a PID or a JID.
  4. 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

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**:

```
unix> ./sdriver.pl -h
Usage: sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
Options:
  -h            Print this message
  -v            Be more verbose
  -t <trace>    Trace file
  -s <shell>    Shell program to test
  -a <args>     Shell arguments
```

-g **Generate output for autograder**

We have also provided 16 trace files (trace{01-16}.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 **trace01.txt** (for instance) by typing:

unix> ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"

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

unix> make test01

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

unix> ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"

or

unix> make rtest01

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). For example:

```
ece373> make test15
./sdriver.pl -t trace15.txt -s ./tsh -a "-p" #
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found.
tsh> ./myspin 10
Job (9721) terminated by signal 2
tsh> ./myspin 3 &
[1] (9723) ./myspin 3 &
tsh> ./myspin 4 &
[2] (9725) ./myspin 4 &
tsh> jobs
[1] (9723) Running ./myspin 3 & [2] (9725) Running ./myspin 4 & tsh> fg %1
Job [1] (9723) stopped by signal 20 tsh> jobs
[1] (9723) Stopped
[2] (9725) Running
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (9723) ./myspin 3 &
tsh> jobs
[1] (9723)
[2] (9725)
tsh> fg %1
tsh> quit
ece373>
```

Hints

-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.

-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).

Evaluation

Your score will be computed out of a maximum of 90 points based on the following distribution:

80 Correctness: 16 trace files at 5 points each.

10 Style points. We expect you to have good comments (5pts) and to check the return value of EVERY system call (5pts).

Your solution shell will be tested for correctness on the class VM, using the same shell driver and trace files that were included in the 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 trace11.txt, trace12.txt, and trace13.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.

Submission Instructions

Put your name, and the name of your partner, at the top of your tsh.c file. Then submit your tsh.c file to Canvas.