CS 324, Winter 2021
Lab Assignment: Writing Your Own Unix Shell, Part 1
Due: Friday, Jan. 29, 11:59PM

Introduction

The purpose of this assignment is to become more familiar with the concepts of process creation, file description duplication for input/output redirection, and pipes.

Logistics

You will work on your own solving the problems for this assignment. The only "hand-in" will be electronic. Any clarifications and revisions to the assignment will be posted on the course Web page.

Hand Out Instructions

Start by downloading the file `shlab-handout1.tar` from the LearningSuite assignment page for this assignment. Save it to the protected directory (the *lab directory*) in which you plan to do your work.

- Type the command `tar xvf shlab-handout1.tar` to expand the tarfile.
- Type the command `make` to compile and link some test routines.
- Type your name and BYU netid 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 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 (which includes lots of comments).

- `eval`: Main routine that parses and interprets the command line. [120 lines]
- `builtin_cmd`: Recognizes and interprets the `quit` built-in command. [10 lines]

Each time you modify your `tsh.c` file, type `make` to recompile it. To run your shell, type `tsh` to 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, in which case the job is referred to as a *pipeline*.

In this lab, a job specified on the command line will always run in the *foreground*, which means that the shell waits for the job to terminate before awaiting the next command line. In the next lab, we will introduce background jobs.

For example, typing the command line

```
tsh> quit
```

causes the shell to simply exit (i.e., with the `exit()`) function. That is a built-in function. Typing the command line

```
tsh> /bin/ls -l -d
```

runs the `ls` program in the foreground—that is, it wait for it to finish before returning the prompt again. By convention, the shell ensures that when the program (i.e., `ls`) 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''.`

Finally, typing the command line

```
    tsh> /bin/ls -l -d | /bin/cat
```

runs the pipeline consisting of a running instance of `ls` sending its standard output sent to the standard input of `/bin/cat`. That is, `echo` and `/bin/cat` are running concurrently, each started with its own set of arguments. This job/pipeline runs in the foreground, until the last process in the pipeline terminates.

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 one or more *commands*, separated by vertical pipe characters (i.e., "`|`"). Each command consists of a `name` and zero or more arguments, all separated by one or more spaces. If `name` for the first command is a built-in command, then `tsh` should handle it immediately and wait for the next command line. Otherwise, `tsh` should assume that `name`—for each command in the pipeline—is the path of an executable file. It will load and run each command in the context of a child process.
- For two consecutive commands in a pipeline, `tsh` connects ("pipes") the standard output of the first command to the standard input of the second command. This is done using Unix pipes and file descriptor duplication. For pipeline consisting of n commands, there should be n child processes connected by $n - 1$ pipes.
- If the less-than or greater-than symbol (i.e., "<" or ">"), is detected on the command line after the arguments for a command, then `tsh` should:
  - treat the word immediately following the symbol as a filename and open it in read mode (for "<") or write mode (for ">");
  - duplicate the file descriptor for the open file onto the file descriptor for stdin (for "<") or stdout (for ">"), which are 0 and 1, respectively.
- For this lab, `tsh` should support just the `quit` built-in command. Others will come in a later lab.

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 12 trace files (`trace{01-03}.txt` and `trace{34-42}.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 stest01
```

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 races. 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:

```
unix> ./sdriver.pl -t trace41.txt -s ./tsh -a "-p"
#
# trace41.txt - Pipeline with stdin/stdout redirection
#
tsh> ./myppid | /bin/grep [0-9] > tshtmp-1-1KEnkI
tsh> ./myppid | /bin/grep [a-z] > tshtmp-2-JmhUC4
tsh> /bin/cat tshtmp-1-1KEnkI
```

```
(2362)
tsh> /bin/cat tshtmp-2-JmhUC4
tsh> /bin/cat < tshtmp-1-1KEnkI | /bin/grep [0-9]
(2362)
tsh> /bin/cat < tshtmp-1-1KEnkI | /bin/grep [a-z]
unix>
```

Shell checker. The `checktsh.pl` program utilizes `sdriver.pl` to execute both your shell and the reference shell and then compare them line by line, so you can see any differences clearly . This tool will be used to automatically *grade* your assignment, so you will want to use it to *check* your assignment! You can use `make` to test a single test case as follows:

```
bass> make test01
./checktsh.pl -v -t trace01.txt

*************************************
* ./checktsh.pl: Checking trace01.txt...
*************************************

./checktsh.pl: Running reference shell on trace01.txt...
#
# trace01.txt - Properly terminate on EOF.
#

./checktsh.pl: Running your shell on trace01.txt...
#
# trace01.txt - Properly terminate on EOF.
#
```

Or you can test all tests:

```
bass> make testall1
./checktsh.pl
Checking trace01.txt...
Checking trace02.txt...
Checking trace03.txt...
Checking trace34.txt...
Checking trace35.txt...
Checking trace36.txt...
Checking trace37.txt...
Checking trace38.txt...
Checking trace39.txt...
Checking trace40.txt...
Checking trace41.txt...
Checking trace42.txt...
```

Hints

- Read every word of Chapter 8 (Exceptional Control Flow) in your textbook.
- The `waitpid`, `fork`, `execve`, `pipe`, `dup2`, and `setpgid` functions will come in very handy. Read the `man` page for each of these functions to find the best way to effectively use them in the shell.
- The `parseline` and `parseargs` functions have been written to help you parse the command line. The former finds all the words on the command line and determines whether or not the job should be run in the background (which applies to the next lab). The latter divides the words into commands (and their respective arguments) in the pipeline, and returns the number of commands in the pipeline. It also identifies the filename(s) (if any) designated by the ">" or "<" symbols for output or input redirection, respectively.

  To help you better understand how these work, here is an example. Suppose the following command-line is provided to your shell:

  ```
  /bin/cat < input.txt | /bin/grep [a-z] > output.txt
  ```

  `parseline()` is passed an array, `char *args[]`, and it is populated thus:

  ```
  args[0] = "/bin/cat";
  args[1] = "<";
  args[2] = "input.txt";
  args[3] = "|";
  args[4] = "/bin/grep";
  args[5] = "[a-z]";
  args[6] = ">";
  args[7] = "output.txt";
  args[8] = NULL;
  ```

  Now, after you pass `args` to `parseargs()`, along with `cmds`, `stdin_redir`, and `stdout_redir` (each of which is an array of `int`), they have the following values.

  - `args` looks the same as it did before, but the pipe and redirection characters have been removed, such that each command, including its arguments, is followed by `NULL`.

    ```
    args[0] = "/bin/cat"
    args[1] = NULL;
    args[2] = "input.txt";
    args[3] = NULL;
    args[4] = "/bin/grep";
    args[5] = "[a-z]";
    args[6] = NULL;
    args[7] = "output.txt";
    args[8] = NULL;
    ```

This will make it easier to pass the args for each command to `execve`. For example, `args[cmds[0]]` can be passed as the args to `execve` for the first command.

  - `cmds` has been populated with as many commands have been identified in the pipeline (two in this case), such that `cmds[0]` contains the index of the first command in `args`, `cmds[1]` contains the index of the second command in `args`, etc.

    ```
    cmds[0] = 0;
    cmds[1] = 4;
    ```

  - `stdin_redir` and `stdout_redir` have been populated with as many commands have been identified in the pipeline (two in this case). In each case, a value greater than 0 contains the index of `args` that contains the filename for which input (`stdin_redir`) or output (`stdout_redir`) should be redirected for the corresponding command in `args`. A value less than 0 indicates that that command has no input or output redirection.

    ```
    stdin_redir[0] = 2;
    stdin_redir[1] = -1;

    stdout_redir[0] = -1;
    stdout_redir[1] = 7;
    ```

- Your shell should save the pid of the child process corresponding to each command in the pipeline. It should then call `waitpid` in each pid, in order, to wait for *all* processes in the pipeline to finish before displaying the prompt (and receiving a new command) again. A simple loop is sufficient for this since the job/pipeline is running in the foreground.

- Remember that file descriptors in a parent process are inherited by their child processes. Thus, any files (or pipes) will need to be opened by the parent in order for them to be appropriately duplicated in the child. This will help you plan the order of your calls to `fork`, `execve`, `pipe`, and `dup2`.

- File descriptors reference file descriptions in a system-wide table. When `close()` is called on a file descriptor, the reference is removed from the file description. Not until *all* references are removed from a file description (i.e., `close()` is called on all file descriptors referencing the description) is that file description removed and considered “closed”. What that means for pipelines is that you must be very careful about closing every descriptor every time it is no longer needed by a process. If you aren’t careful, some will be left open, and you will find that your pipeline hangs.

- All commands in a pipeline should be in the same progress group, but that process group should be *different* than the process group of the shell. The group ID should be the process ID of the *first* command in the pipeline. Thus, for every command in a pipeline, after `fork` is called, the parent process should call `setpgid(pid, pgid)`, where `pid` is the process ID of the child process that was just created and `pgid` is the process ID of the *first* child process in the pipeline.

  Now the collective child processes corresponding to the commands in a single pipeline are associated with a single process group ID. This will help in the next lab where a signals will be handled and need to go to an entire pipeline of commands comprising a job, not just a single process.

- 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.
- 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`.

Evaluation

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

96 Correctness: 8 trace files at 12 points each.

4 Compiles without warnings.

Your solution shell will be tested for correctness on a Linux machine, using the same shell driver and trace files that were included in your lab directory. Your shell should produce **identical** output on these traces as the reference shell, with the following exceptions:

- The PIDs can (and will) be different.
- The names of temporary files (created on-the-fly for testing) will be different.

Hand In Instructions

Important: Copy your `tsh.c` file to one of the CS lab machines, and ensure your file compiles as expected (i.e., using `make`) and that *all* the test cases run as expected (e.g., with `make testall1`). You can use `scp` or `sftp` to transfer your files.

To submit your work, upload (only) your *working* `tsh.c` to the assignment page on LearningSuite.

Good luck!