CMPS 6100 Lab 08

In this lab, you will build your own tiny shell.

Refer back to the README.md for instruction on git and how to submit properly to get all the points you've earned.

Introduction

The purpose of this lab is for you to become more familiar with the concepts of *processes*, the `fork()` and `exec` system calls, parent and child processes, and signalling.

You'll do this by writing a simple Unix shell program, the "tiny shell", or *tsh*, that supports job control.

In addition to learning about many important systems topics, you will also get much more experience writing code in C.

Instructions

You are provided with an initial shell program which you will modify.

Start by logging into `cmps-lab.cs.tulane.edu` as usual and clone this repository into your `labs` directory.

Once you have cloned the repository, you can get started with familiarizing yourself with the tiny shell by doing the following

- `cd` into the `src` directory
- use the command `make` to compile some test programs
- use `vi` (or your editor of choice) to put your name 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 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`**: The major function which parses and interprets the command line. [70 lines]
- **`parseline`**: Split a command string into its arguments and save them in `argv`. Return 1 if the command should be executed in the backgroun, 0 if it should be executed in the foreground. (You've already done this!)
- **`builtin_cmd`**: Recognize and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs` [25 lines]
- **`sigint_handler`**: Handles **SIGINT** (*ctrl-c*) signals. [15 lines]

Every time you modify your `tsh.c` file, use the `make` command to recompile it. To run your shell, enter `tsh` on the command line:

```
cmps-lab:~ $ ./tsh
tsh> type commands to your shell here
```

General Overview of Unix Shells

> This section is reproduced from the previous lab document "What is a shell?" I recommend rereading this in preparation of your implementation.

As mentioned, a shell is the most powerful interface we have to computers. GUI and (now-a-days) voice interfaces are limited by what they are programmed to do. The shell is only limited by the capabilities of the computer itself.

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, entering the command:

```
$ jobs
```

causes the shell to execute the buit-in `jobs` command. Entering the command:

```
$ sleep 5
```

runs the sleep program for 5 seconds in the foreground. This causes a process to sleep for 5 seconds before continuing. `5` is a command line argument that modifies the behavior of `sleep`. Alternatively entering the command:

```
$ sleep 10 &
```

runs the sleep program in the background. At this point, if you run `$ jobs` you will see it listed!

Shell Commands

When a command is entered on the command line, by convention it is passed into its programs main function as an array of strings.

For example, the shell will pass the following command

```
$ /bin/ls -l -a
```

into

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

as:

- `argc == 3`
- `argv[0] == "/bin/ls"`
- `argv[1] == "-l"`
- `argv[2] == "-a"`

Signals and Job Control

The OS uses signals to interrupt and control jobs. Unix shells use signals to implement the notion of *job control*, allowing 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, if you hit `ctrl-c` on a running job in the foreground of your shell, you cause a `SIGINT` ("signal interrupt") to be sent to each process in that job. The default action for `SIGINT` is to terminal 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 foreground job
- `kill <job>`: Terminate a job

The `tsh` Specification

Your `tsh` 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 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.
- `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.
- Program output can be redirected to a file using the standard form (`> output-filename`), and program input can be redirected from a file using the same form (`< input-filename`). (Pipes (`|`) are *not* required for this assignment.)
- Typing `ctrl-c` or `ctrl-z` should cause signals `SIGINT` or `SIGTSTP` respectively 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.
- 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.)

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`:

```
systems-lab$ ./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 20 trace files (`trace{01-20}.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:

```
cmps-lab$ ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
```

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

```
cmps-lab$ make test01
```

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

```
cmps-lab$ ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
```

or

```
cmps-lab$ make rtest01
```

Helpful tip from a previous student: To see exactly how your shell output differs from the reference shell for a specific test, use this cool trick (a bash-specific trick known as process substitution):

```
cmps-lab$ diff -u <(make test07) <(make rtest07)
```

This will show you how your shell's output from `test07` different from the reference shell. Hopefully this will help you find bugs.

Finally, you can check all of the tests with:

```
cmps-lab$ ./checktsh.py
```

This will notify you of any test failures.

Hints

- 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 `fork`, `kill`, `execv`, `setpgid`, and `sigprocmask` functions will come in very handy.
  - Note that `kill` here refers to the system call `kill` (manual accessed with the command `man 2 kill`), not the *bash* built-in function `kill` (manual accessed with the command `man kill` or `man 1 kill`)
- When you implement your `sigint_handler`, be sure to send the `SIGINT` signal 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.

- 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`.

  - Learning how to use `sigprocmask` is one of the tricky parts of this lab. Do not be afraid to research `sigprocmask` and how to use it.

- 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 `execv`, 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).

- Learning about the various concepts, mechnanisms, and calls required to implement your shell is a major part of this lab. Upon first reading this document, you may feel out of element. It's ok. You got this. Revisit this document as you continue to work through implementing your shell, and it will start to make more sense as you go. You will find insights that will help you overcome issues as you encounter them.

  - For the necessary context for the lab, refer back to *Ch 5 Interlude: Process API* from *Operating Systems: Three Easy Pieces*.

  - For information about the various functions and system calls referred to and required by this lab, don't be afraid to look up their documentation and examples of their use. You can access the manual for signals in general by `man 7 signal`, for `sigprocmask` by `man sigprocmask`, and so on for other system calls as well.

Dealing with Segmentation Faults

Is your program crashing with a `Segmentation Fault`? This is often due to a null pointer reference involving strings, or more generally a memory error relating to the misuse of pointers. Fortunately there is a good tool to find out exactly where the problem lies in your code: `gdb`

I have a program that I know crashes: `foo_bar`. I compiled it with debugging symbols enabled.

If I run it, I see:

```
$ ./foo_bar
Segmentation fault
```

To debug:

```
$ gdb ./foo_bar
Reading symbols from ./foo_bar...
(gdb)
```

You are given the prompt `(gdb)`. Enter the command `run` to execute your program.

```
(gdb) run
Starting program: /home/amaus/foo_bar

Program received signal SIGSEGV, Segmentation fault.
__strlen_evex () at ../sysdeps/x86_64/multiarch/strlen-evex.S:77
77  ../sysdeps/x86_64/multiarch/strlen-evex.S: No such file or directory.
(gdb)
```

To find out where the SEGFAULT was generated from, issue the command `backtrace`:

```
(gdb) backtrace
#0  __strlen_evex () at ../sysdeps/x86_64/multiarch/strlen-evex.S:77
#1  0x00007ffff7e58f76 in __vfprintf_internal (s=0x7ffff7fc06a0 <_IO_2_1_stdout_>,
    format=0x555555556004 "%s\n", ap=ap@entry=0x7fffffffe250,
    mode_flags=mode_flags@entry=0) at vfprintf-internal.c:1688
#2  0x00007ffff7e43d9b in __printf (format=<optimized out>) at printf.c:33
#3  0x000055555555517c in read () at foo_bar.c:6
#4  0x0000555555555198 in main (argc=1, argv=0x7fffffffe458) at foo_bar.c:11
(gdb)
```

Look for the highest line that refers to your source file and that shows you which line caused the SEGFAULT to be issued. In this case, line 6. In your program, investigate that line and those before it that relate to it. This will help you identify and fix the issue.

> `gdb` is able to see exactly which line caused the SEGFAULT only if you comile your programs using the `-g` option (as in `$ gcc -g foo_bar.c -i  foo_bar`)

To quit gdb:

```
(gdb) quit
A debugging session is active.

    Inferior 1 [process 103285] will be killed.

Quit anyway? (y or n) y
```

`foo_bar.c` is included in this repository.

Resources

- Harvard's CS50 Manual Page
- Harvard's CS50 Duck AI
- Stanford's Essential C