Homework 2: Shell

Due: March 3, 2022

Contents

1 Overview

In this homework, you'll be building a shell, similar to the `bash` shell you use on your CS 162 Virtual Machine. When you open a terminal window on your computer, you are running a shell program, which is `bash` on your VM. The purpose of a shell is to provide an interface for users to access an operating system's services, which include file and process management. `sh` (Bourne shell) is the original Unix shell, and there are many different flavors of shells available. Some other examples include `ksh` (Korn shell), `tcsh` (TENEX C shell), and `zsh` (Z shell). Shells can be interactive or non-interactive. For instance, you are using bash non-interactively when you run a bash script. bash is interactive when invoked without arguments, or when the `-i` flag is explicitly provided. The operating system kernel provides well-documented interfaces for building shells. By building your own, you'll become more familiar with these interfaces and you'll probably learn more about other shells as well.

1.1 Getting Started

To get started, log in to your development environment and get the starter code.

```
> cd ~/code/personal/
> git pull staff master
> cd hw-shell
```

We have added starter code for your shell which includes a string tokenizer that splits strings into words. To run the shell,

```
> cd ~/code/personal/hw-shell
> make
> ./shell
```

2 Directory

The skeleton code for your shell has a **dispatcher** for "built-in" commands. Every shell needs to support a number of built-in commands, which are functions in the shell itself, not external programs. For example, the `exit` command needs to be implemented as a built-in command, because it exits the shell itself. So far, the only two built-ins supported are `?`, which brings up the help menu, and `exit`, which exits the shell.

Add a new built-in `pwd` that prints the current working directory to standard output. Then, add a new built-in `cd` that takes one argument, a directory path, and changes the current working directory to that directory.

You may find the syscalls `chdir` and `getcwd` helpful. Check out `man` pages for usage details.

3 Program Execution

If you try to type something into your shell that isn't a built-in command, you'll get a message that the shell doesn't know how to execute programs. Modify your shell so that it can execute programs when they are entered into the shell. The first word of the command is the name of the program. The rest of the words are the command-line arguments to the program.

You can assume that the first word of the command will be **the full path to the program**. So instead of running `wc`, you would have to run `/usr/bin/wc`. In the following parts, you will implement support for simple program names like `wc`. However, you can pass some autograder tests by only supporting full paths.

You should use the functions given in `tokenizer.h` for separating the input text into words. You do not need to support any parsing features that are not supported by `tokenizer.h`.

When your shell needs to execute a program, it should `fork` a child process, which calls one of the functions from the `exec` family to run the new program. **You may not use `execvp`** (see Path Resolution). The parent process should wait until the child process completes and then continue listening for more commands.

Once you implement this funcitionality, you should be able to execute program.

```
> ./shell
0: /usr/bin/wc shell.c
      77     262    1843 shell.c
1: exit
```

4 Path Resolution

You probably found that it was a pain to test your shell in the previous part because you had to type the full path of every program. Luckily, every program (including your `shell` program) has access to a set of **environment variables**, which is structured as a hashtable of string keys to string values. One of these environment variables is the `PATH` variable. You can print the `PATH` variable of your development environment.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:...
```

When `bash` or any other shell executes a program like `wc`, it looks for a program called `wc` in each directory listed in the `PATH` environment variable and runs the first one that it finds. The directories in `PATH` are separated with a colon.

Modify your shell so that it uses the `PATH` variable from the environment to resolve program names. Typing in the full pathname of the executable should still be supported. **Do not use `execvp` since the autograder looks for `execvp`, and you won't receive a grade if that word is found.** Use `execv` instead and implement your own `PATH` resolution.

5 Redirection

When running programs, it is useful to provide input from a file or to direct output to a file. The syntax `[process] > [file]` tells your shell to redirect the process's standard output to a file. Similarly, the syntax `[process] < [file]` tells your shell to feed the contents of a file to the process's standard input.

Modfiy your shell so that it supports redirecting stdout and stdin to/from files. You do not need to support redirection for shell built-in commands. You do not need to support stderr redirection or appending to files (e.g. `[process] >> [file]`). You can assume that there will always be spaces around special characters `<` and `>`. Be aware that `< [file]` or `> [file]` are not passed as arguments to the program (i.e. not in `argv`).

6 Pipes

Other times, it is useful to provide the output of a program as the input of another program. The syntax `[process A] |[process B]` tells your shell to pipe the output of process A to the input of process B. In other words, the output of program A becomes the input of program B.

Modify your shell so that it supports pipes between programs. You can assume there will always be spaces around the special character |.

For an optional challenge, you may try to pipe more than once, as seen in the example below. For example:

```
[process A] | [process B] | [process C]
```

There will be autograder tests worth 0 points if you would like to test out your functionality for piping between multiple processes.

7 Signal Handling

Most shells let you stop or pause processes with special keystrokes. These special keystrokes, such as `Ctrl-C` or `Ctrl-Z`, work by sending signals to the shell's subprocesses. For example, pressing `CTRL-C` sends the `SIGINT` signal, which usually stops the current program; pressing `CTRL-Z` sends the `SIGTSTP` signal, which usually sends the current program to the background. Recall that your terminal window is running a shell program itself. The shell must make sure that these keystrokes do not stop the shell program itself. If you try these keystrokes in your homemade shell right now, the signals are sent directly to the shell process itself. This means that attempting to `CTRL-Z` a subprocess of your shell, for example, will also stop the shell itself. We want to have the signals affect only the subprocesses that our shell creates.

7.1 Example: Shells in Shells

On your Vagrant VM, you'll be executing a short series of commands in order to better understand the correct behavior. We'll primarily be making use of two commands, `ps` and `jobs`. Recall that `ps` gives you information about all processes running on the system, while `jobs` gives you a list of jobs that the current shell is managing. Enter the following commands in your terminal, and you should see similar behavior:

```
$ ps
PID TTY           TIME CMD
20970 ttys002    0:01.30 -bash
$ sh
sh-3.2$ ps
PID TTY           TIME CMD
20970 ttys002    0:00.63 -bash
22323 ttys004    0:00.01 sh
```

At this point, we have started a `sh` shell within our `bash` shell.

```
sh-3.2$ cat
hello
hello
world
world
^Z
[1]+  Stopped(SIGTSTP)        cat
sh-3.2$ ps
PID TTY           TIME CMD
20970 ttys004    0:00.63 -bash
22323 ttys004    0:00.02 sh
22328 ttys004    0:00.01 cat
```

Notice how sending a `CTRL-Z` while the `cat` program was running did not suspend the `sh` nor the `bash` programs.

```
sh-3.2$ jobs
[1]+  Stopped(SIGTSTP)        cat
sh-3.2$ exit
$ ps
PID TTY           TIME CMD
20970 ttys004    0:00.65 -bash
```

Since `exit` terminates the shell, we terminated the `sh` program. Enter `exit` again and your terminal will close.

Before we explain how you can achieve this effect, let's discuss some more operating system concepts.

7.2 Process Groups

We have already established that every process has a unique process ID (pid). Every process also has a (possibly non-unique) process group ID (pgid) which, by default, is the same as the pgid of its parent process. Processes can get and set their process group ID with `getpgid()`, `setpgid()`, `getpgrp()`, or `setpgrp()`[1].

Keep in mind that when your shell starts a new program, that program might require multiple processes to function correctly. All of these processes will inherit the same process group ID of the original process. So, it may be a good idea to put each shell subprocess in its own process group, to simplify your bookkeeping. When you move each subprocess into its own process group, the pgid should be equal to the pid.

7.3 Foreground Terminal

Every terminal has an associated "foreground" process group ID. When you type `CTRL-C`, your terminal sends a signal to every process inside the foreground process group. You can change which process group is in the foreground of a terminal with "`tcsetpgrp(int fd, pid_t pgrp)`". The `fd` should be `0` for "standard input".

7.4 Overview of Signals

Signals are asynchronous messages that are delivered to processes. They are identified by their signal number, but they also have somewhat human-friendly names that all start with `SIG`. Some common ones include:

`SIGINT`
: Delivered when you type `CTRL-C`. By default, this stops the program.

`SIGQUIT`
: Delivered when you type `CTRL-`. By default, this also stops the program, but programs treat this signal more seriously than `SIGINT`. This signal also attempts to produce a core dump of the program before exiting.

`SIGKILL`
: There is no keyboard shortcut for this. This signal stops the program forcibly and cannot be overridden by the program.

`SIGTERM`
: There is no keyboard shortcut for this either. It behaves the same way as `SIGQUIT`.

`SIGTSTP`
: Delivered when you type `CTRL-Z`. By default, this pauses the program. In bash, if you type `CTRL-Z`, the current program will be paused and bash (which can detect that you paused the current program) will start accepting more commands.

`SIGCONT`
: Delivered when you run `fg` or `fg %NUMBER` in bash. This signal resumes a paused program.

`SIGTTIN`
: Delivered to a background process that is trying to read input from the keyboard. By default, this pauses the program, since background processes cannot read input from the keyboard. When you resume the background process with `SIGCONT` and put it in the foreground, it can try to read input from the keyboard again.

`SIGTTOU`
: Delivered to a background process that is trying to write output to the terminal console, but there is another foreground process that is using the terminal. Behaves the same as `SIGTTIN` by default.

[1] Again, see the `man` pages for more information.

In your shell, you can use `kill -XXX PID`, where `XXX` is the human-friendly suffix of the desired signal, to send any signal to the process with process id `PID`. For example, `kill -TERM PID` sends a `SIGTERM` to the process with process id `PID`.

In C, you can use the `sigaction` system call to change how signals are handled by the current process. The shell should basically ignore most of these signals, whereas the shell's subprocesses should respond with the default action. For example, the shell should ignore `SIGTTOU`, but the subprocesses should not. **Beware:** forked processes will inherit the signal handlers of the original process. Reading `man 2 sigaction` and `man 7 signal` will provide more information. Be sure to check out the `SIG_DFL` and `SIG_IGN` constants. For more information on process group and terminal signaling, please go through this tutorial.

Your task is to ensure that each program you start is in its own process group. When you start a process, its process group should be placed in the foreground. Stopping signals should only affect the foregrounded program, not the backgrounded shell.

8 Foreground and Background Processes

This part is optional. There will be tests on the autograder if you would like to test your functionality, but they will be worth 0 points.

8.1 Background Processes

So far, your shell waits for each program to finish before starting the next one. Many shells allow you run a command in the background by putting an `&` at the end of the command line. After the background program is started, the shell allows you to start more processes without waiting for the background process to finish.

Modify your shell so that it runs commands that end in an `&` in the background. Once you've implemented this feature, you should be able to run programs in the background with a command such as `/bin/ls &`.

You should also add a new built-in command `wait`, which waits until *all* background jobs have terminated before returning to the prompt.

You can assume that there will always be spaces around the `&` character. You can assume that, if there is a `&` character, it will be the last token on that line.

8.2 Foreground/Background Switching

Most shells allow for running processes to be toggled between running in the foreground versus in the background. You can optionally add two built-in commands to support this:

`fg [pid]` Move the process with id `pid` to the foreground. The process should resume if it was paused. If `pid` is not specified, then move the most recently launched process to the foreground.

`bg [pid]` Resume a paused background process. If `pid` is not specified, then resume the most recently launched process.

You should keep a list of all existing processes (whether they are in the foreground or background) and their `pid`s. Inside this list, you should also keep a `struct termios` to store the terminal settings of each program.

9 Submission

To submit and push to the autograder, push your changes to your repo which should trigger the autograder unless you're using slip days in which case you need to manually run it. Within a few minutes you should receive an email from the autograder. If you don't receive an email from the autograder within half an hour, please notify staff via a private post on Piazza. Your code should not include extraneous or debugging print statements as this will interefere with the autograder. Your written responses should be submitted to the Gradescope and will not be graded by the autograder.