HW 1: Shell

CS 162

Due: September 21, 2015

Contents

The shell (e.g., bash) is a program that is so closely associated with the operating system, that most people think of it as part of the operating system. But in reality, the operating system provides a clean interface for accessing resources, while the shell only uses this interface to provide the ability to run programs. You can use these same interface to build your own shell. This will give you a better sense of the user-system interface used to access these resources. This assignment will be due **September 21, 2015 at 11:59 pm PDT**.

1 Getting started

Log in to your Vagrant Virtual Machine and run:

```
cd ~/code/personal
git pull staff master
cd hw1
```

We have added starter code for your shell and a simple Makefile in the `hw1` directory. A text parser (good enough to help you pass the autograder tests) and helper functions for IO operations have been included. In order to run the shell:

```
make
./shell
```

In order to terminate the shell after it starts, either type `quit` or press `CTRL-D`.

2 Add support for cd and pwd

The starter code for your shell has a dispatcher to support 'built-ins'. This dispatch pattern shows up frequently in operating systems; for example, it appears in the syscall vector to direct syscalls to the appropriate kernel handler and in the interrupt vector to direct interrupts to the appropriate interrupt handler. Here we look up built-in commands in a built-in commands vector to transfer control to a built-in command handler. So far, the only two built-ins supported are `?`, which brings up the help menu, and `quit`, which exits from the shell.

Programs normally access operating capabilities through the Standard C Library, libc. See, for example, http://www.gnu.org/software/libc/manual/pdf/libc.pdf. **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 name, and changes the current working directory to that directory.**

Check in your solution to this part. In your VM:

```
git add .
git commit -m "Finished adding basic functionality into the shell."
git push personal master
```

3 Execute programs

If you try to type anything into your shell that is not a valid built-in, you get a message that the shell doesn't know how to execute programs. Modify your shell so that, when it receives a command that isn't a known built-in, it forks a child process to execute the command as a program. The first word of the command should be the program's name. The rest of the words are the command-line arguments to the program.

For this step, 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 run `/usr/bin/wc`. In the next section, you will implement support for looking up program names like `wc`. But you can pass some tests without supporting that feature.

You can use the functions defined in `parse.c` for parsing command line input. You do not need to support any parsing features that are not supported by `parse.c`. Specifically, you do not need to support quotes, escape sequences, variable interpolation, or command interpolation. For example:

```
vagrant@development [00:00:00] hw1 $ ./shell
0: /usr/bin/wc shell.c
      77     262    1843 shell.c
1: quit
vagrant@development [00:00:00] hw1 $
```

Your textbook provides some suggestions about how to implement this. Basically, when your shell receives a program to execute, it should fork a child process, which calls a variant of `exec` to run the new program. The parent process should wait until the child process completes and then go back to listening for more commands.

Check in your solution to this part. In your VM:

```
git add .
git commit -m "Finished creating child process to executes files."
git push personal master
```

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 pathname of every executable. Operating systems provide an "environment" that functions like a hashtable of string to strings. One of these environment entries is the `PATH` variable. You can print the `PATH` variable of your current environment on your Vagrant VM: (use bash for this, not your homemade shell!)

```
vagrant@development [00:00:00] hw1 $ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:...
```

When a shell executes a program like `wc`, it looks for that program in each directory on the path and executes the first one that it finds. The directories on the path are separated with a colon. This process is called resolving the path.

Modify your shell so that it reads the `PATH` variable from the environment and uses it to resolve program names in the path. Typing in the full pathname of the executable should still be supported. *Do not use* `execvp`. *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.* Test your work and commit it.

5 Input/Output Redirection

When running programs, it is sometimes useful to be able to feed in input from a file or to write the output to a file. The syntax `[process] > [file]` specifies to write the process's output to the file. Similarly, the syntax `[process] < [file]` specifies to feed in the contents of the file into the process. The commands `dup2` and `strcmp` may be useful here.

Modfiy your shell so that it supports redirecting stdin and stdout to 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 `>`.

6 Signal Handling and Terminal Control

Shells let you interrupt or pause processes with special key strokes. For example, pressing `CTRL-C` sends `SIGINT` which stops the current program, while pressing `CTRL-Z` sends `SIGTSTP` which sends the current program to the background. Right now, these signals are sent directly to our shell. This is not what we want, since for example attempting to `CTRL-Z` a process will also stop our shell. We want to have the signals affect the processes that our shell creates, not the shell itself.

6.1 Process groups

Every process has a process ID (pid). Every process also has a process group ID (pgid), which by default, is the same as the pgid of its parent. Processes can get/set their process group ID with `getpgid`, `setpgid`, `getpgrp`, or `setpgrp`.

6.2 Foreground terminal

Every terminal has a 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 the foreground process group ID of a terminal with `tcsetpgrp(int fd, pid_t pgrp)`. The `fd` should be `0` for "standard input", which is connected to your terminal. (Standard output or error would also work, but input makes the most sense.)

6.3 Overview of signals

Signals are asynchronous messages that are delivered to processes. They are identified by their signal number, but they also have 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. (Most other signals can be ignored 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 background processes that are 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 background processes that are 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.

You can use `kill -TERM PID` to send `SIGTERM` to the process with process id `PID`.

6.4 Suggested implementation

You can use the `signal` function to change how signals are handled by the current process. The shell should basically ignore most of these signals, whereas the programs it runs should do the default action. Beware: forked processes will inherit the signal handlers of the original process. Reading `man 2 signal` and `man 7 signal` will provide more information. For more information about how signals work, please work through the tutorial here.

You should 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. You can use the function we provided, `put_process_in_foreground`, to do this, or you can call `tcsetpgrp` directly.

7 Background processing

Our shell so far runs each command to completion before allowing you to start the next. 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 background process to finish.

Modify your shell so that it runs commands that end in an "&" in the background. You only need to support backgrounding for programs, not built-in commands. For example,

```
/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 Optional: Foreground/Background Switching

Most shells allow for running processes to be toggled between running in the foreground versus in background. Please add two additional built-ins 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.

8.1 Suggested implementation

You should ensure that each program you start is in its own process group. The process group id of a program should be the same as its pid. You should keep a list of all programs you've started, whether they are in the foreground or background. Inside this list, you should also keep a `struct termios tmodes` to store the terminal settings of each program, so you can restore them with `put_process_in_foreground`.

9 Optional: Terminal modes

Each terminal has a bunch of settings that can be configured by programs. Some of these settings are: do characters show up when you type (they don't when you run sudo), and what should happen when you press `CTRL-C`. Read `man termios` for more details.

When you run a program, it will change a bunch of terminal settings. When the program finishes, it usually will NOT change them back. So, you need to save a copy of these settings if you want to restore them. We have provided `put_process_in_foreground` and `put_process_in_background` that should abstract away saving/restoring terminal modes for you.

10 Autograder and Submission

To submit and push to autograder, first commit your changes, then do:

```
git push personal master
```

Within 30 minutes you should receive an email from the autograder. (If you haven't received an email within half an hour, please notify the instructors via a private post on Piazza.)

Please ensure that your solution does not print any extraneous output when stdin is not a terminal. That is, any time a built-in or a process in run with your shell, only the *output* of the built-in or process should be printed. Please do not print anything extra for debugging, as this can mess up your autograder results.