HW 1: Initial Shell

David Culler

August 31, 2014

Contents

1 Setup

The shell, e.g., bash, csh, or sh, is an application program that is so closely associated with the operating system that most people think of it as part of the OS. But really the OS provides a clean abstraction for accessing resources and manages sharing of those resources. The shell provides a command interpreter and the ability to run programs on the OS. (Your starter shell gives a little sense of this by getting and printing its process id (PID) along with that of its parent - a real shell.) In building one, you will get a better sense of the user/system interface than you do from more typical applications.

In your vagrant vm

```
cd cs162-hw
git pull staff master
cd hw1
```

You will find starter code shell.c and a simple Makefile. You will notice the use of ".h" files to provide a rough C approximation to classes. A parser and a file for io operations have been included as well.

In order to run the shell:

```
make
./shell
```

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

2 Using libc in the shell

The skeleton shell has a dispatcher to support 'builtins'. This dispatch pattern shows up frequently in operating systems; for example, it appears in vectoring syscalls to the appropriate kernel handler and in vectoring interrupts to the interrupt handler. Here we do a look up to transfer control to a command handler. So far the only two builtins 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. To warm up, let's make this shell a little more interesting. Currently the prompt is just the command line number. Modify this to include the current working directory (see man getcwd) in the prompt. Add a command a new built-in 'cd' that changes the current working directory. Test your program on all the relevant cases (and fix any bugs you may find along the way.)

Check in your solution to this part. In your vagrant vm

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

3 A simple shell with Exec

You will notice that anything you type that is not a valid built-in results in a message that it doesn't know how to exec programs. Extend your shell of part 1 to fork a child process to execute the command passing it the command line argument. For example:

```
culler@dhcp-45-107:~/Classes/cs162/fa14/cs162git/ta/hw1$ ./shell
./shell running as PID 21799 under 17720
1 /Users/culler/Classes/cs162/fa14/cs162git/ta/hw1: /usr/bin/wc shell.c
```

```
      77     262    1843 shell.c
2 /Users/culler/Classes/cs162/fa14/cs162git/ta/hw1: quit
Bye
```

Your book provides a rough guideline. Your shell should fork a child process which execs the executable file. The parent shell process should wait until the subprocess completes.

Check in your solution to this part. In your vagrant 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 rather a pain to test your shell in the previous part because you had to type the full pathname of every executable. Most operating systems provide an "environment" in which to resolve various names to their values. For example

```
echo $PATH
```

prints the search path that the shell uses to locate executables. It looks for the file in each directory on the path, separated by ":" and executes the first one that it finds. This process is called resolving the path.

Modify your shell to access the PATH variable from the environment and use it to resolve executable file names. Do not use `execvp`. Test your work and commit it.

5 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. The shell responds with the prompt and allows you to start more processes.

Modify your shell so that it runs commands that are terminated by an "&" in the background. Backgrounding should be ignored for built-ins. Add a new builtin, `wait`; it should wait until all backgrounded jobs have terminated before returning to the prompt.

Test your work and commit the changes.

6 Bonus

At this point you may be feeling, "hey, what's so special about bash, I could write my own shell!" We have left off a lot of functionality. We don't have pipes. We haven't redirected stdin and stdout. We don't have an interpreter. We don't have an environment. We don't have the sense not to issue a prompt if we are not running from a terminal. Oh well. We'll put some of these in later. If you feel inspired, feel free to expand the shell that you have built. Tell us what you decided to add and why.

7 Autograder & Submission

To push to autograder do:

```
git add .
git commit -m "hw1 test"
git checkout -b ag/hw1
git push personal ag/hw1
```

Within a few minutes you should receive an email from the autograder. (If not, please notify the instructors via Piazza).

Now in order to finally submit your code, you need to push to the branch `release`

```
make clean
git add .
git commit -m "hw1 submission"
git checkout -b release/hw1
git push personal release/hw1
```

The reason we gave you two types of branches with an autograder, is that the `ag/*` are testing branches, nothing on it will be graded whereas you **must** submit to release in order to get graded. So please only push to `release/*` when you intend to submit.