HW 1: Initial Shell

Due: February 09, 2015

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 code/personal
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 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. You may find the functions defined in parse.c to be useful for parsing command line input. In particular, you do not need to support delimiters not defined in parse.c (eg. quotes, escape characters). For example:

```
kubi@dhcp-45-107:~/Classes/cs162/sp15/cs162git/ta/hw1$ ./shell
./shell running as PID 21799 under 17720
1 /Users/kubi/Classes/cs162/sp15/cs162git/ta/hw1: /usr/bin/wc shell.c
      77     262    1843 shell.c
2 /Users/kubi/Classes/cs162/sp15/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. Typing in the full pathname of the executable should still be supported. Do not use `execvp`. 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 to a file. The syntax `[process] > [file]` specifies to write the process' 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 `strstr` may be useful here.

Modfiy your shell so that we support redirecting stdin and stdout. Test your work and commit the changes.

6 Process Bookkeeping

Not only do we want our shell to run programs, but we want it to keep track of programs that are currently running, programs that have either been stopped or have terminated, and whether programs hold the terminal or not.

Modify your shell so that whenever your shell runs a program, it creates a `process` struct and fills in its appropriate fields. We have defined the `process` struct that you will be using for you.

Notice that we are not exactly changing the functionality of the shell at the moment, but this is necessary for supporting background processes and process control, as you will implement below.

You also want to make sure that when a process is stopped or is completed, that its data structure is updated to reflect that.

Test your work and commit it.

7 Signal Handling

To interrupt/pause/stop running processes we will use glibc's signals. For example, `CTRL-C` sends `SIGKILL` and `CTRL-Z` sends `SIGSTOP`. However, since our shell is running inside another shell, by default these signals are sent to our shell. This is not what we want, since for example attempting to stop a process will also stop our shell. We want to "ignore" the signals inside our shell's process, and reenable them for the processes that our shell spawns. Reading the linux man page for signal may be useful here.

You must also ensure that each process lies in its own process group. A process group is a collection of one or more processes. Process groups are used to help control signal distribution: a signal that is directed towards a process group is sent to all members of the group. We want to ensure that when our shell spawns multiple processes, that if we send for example `SIGSTOP`, then we only stop the foreground process. You may find the function `setpgid` to be useful.

Test your work and commit it.

8 Foreground/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. For example, both

`ls&`, and

`ls &`

runs the ls command in the background. Backgrounding should be ignored for built-ins. Add three new builtins:

`wait`; wait until all backgrounded jobs have terminated before returning to the prompt.

`fg [pid]`; Move the process with id `pid` to the foreground. If `pid` is not specified, then move the most recently launched process to the foreground.

`bg [pid]`; Move the process with id `pid` to the background. If `pid` is not specified, then move the most recently launched process to the background.

Additionally, if a process is in the foreground, we want that process to have control of the terminal. Please modify your shell so that if a process is in the foreground, that process has control of the terminal. When a process exits, then the shell regains control of the terminal. You may find the code already defined in `init_shell` to be useful. You may also find the documentation for `tcsetpgrp` to be useful.

Test your work and commit the changes.

9 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.