CS 232 Operating Systems

A Command Shell

Overview

A *shell* is a program that repeatedly reads a user command from the keyboard and executes that command on behalf of the user. You might think of the shell as a *chat* program that allows you to ask the operating system to perform tasks for you. How cool is that?!

When you start up your shell, it prints out a prompt, which shows the "current working directory" or "cwd", followed by "$ ". The program then awaits a command:

```
/home/vtn2/proj/shell/$
```

When the user types a command, the program should perform that command and then prompt for the next one:

```
/home/vtn2/proj/shell/$ ps
[Child pid: 2485]
     PID TTY          TIME CMD
     2208 pts/2    00:00:00 bash
     2428 pts/2    00:00:00 xyshell
     2485 pts/2    00:00:00 ps
[2485 -> 0]
/home/vtn2/proj/shell/$
```

If the user types a command with an ampersand, the shell runs the command "in the background". This means the shell does not wait for the command to complete before prompting for the next command:

```
/home/vtn2/proj/shell/$ sleep 5 &
[Child pid: 2493]
/home/vtn2/proj/shell/$
```
[the sleep command completes 5 seconds later]

There are two kinds of commands you can run: built-in commands and executables that have been installed on the computer. The built-in command you must implement are:

`exit`	The shell program exits

`cd` **`directory`**	Change the current working directory to the given **directory**. Do this by calling the `chdir()` system call.
	Your prompt should reflect the change to the new directory.
	You do not have to support anything fancy like:
	`cd`
	`cd ~`
	`cd -`

`pwd`	print the current working directory

For non-built-in commands, your program needs to do what bash, ksh, tcsh, csh, zsh, or sh do: use the PATH environment variable to find where to look for the command to execute. The PATH environment variable commonly looks something like this:

```
PATH=/home/vtn2/.local/bin:/home/vtn2/bin:bin:/usr/local/sbin:/usr/local/bin:
/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

As you can see, the path is a colon-separated list of directories that may contain executable programs. When you run a program in the shell, each directory in the PATH is searched for the executable. If it is found, the program is run. If it is not found in any directory, an error is printed.

Example: suppose you run `ps`. The shell first looks for `/home/vtn2/.local/bin/ps`. Then it looks for `/home/vtn2/bin/ps`, then `bin/ps`, then `/usr/local/sbin/ps`, etc. Eventually it finds `ps` in `/bin` and runs `/bin/ps`.

(Note that the current working directory (denoted by .) is not found in the PATH. This is why you cannot run programs straight out of your current directory unless you explicitly say *./command.)*

The other way to run a command is to specify its absolute path: do this by specifying the full path to the executable. E.g., running `/bin/ps` will not search the path since the full pathname was specified. An alternative is to give ./command to run a program out of the current directory.

Note that commands can take multiple arguments. E.g.,

```
ls -1 -F /tmp
```

This runs the `ls` command with the `-1` and `-F` arguments on the `/tmp` directory. Arguments are separated by one or more spaces.

As noted above, you also have to support having a `&` on the end of a command:

```
sleep 5 &
```

The ampersand indicates that the command should be run in the background. The ampersand is not passed as an argument to the command. Also, only one ampersand can be given and if it is given, it must be the last "word" on the line.

Recommended Plan of Attack

Start by Accepting this assignment from GitHub Classroom. Your calvin login should already be in the list of registered students. If it isn't, just go on -- you'll be added to the classroom. When you are done, get the id of your repo and use `git clone <the-id>` to get a new repo. Note: there is no code in the repo -- you have to add everything.

There are 4 challenging parts to this assignment: 1) reading and parsing the PATH environment variable, 2) reading, parsing, and error-checking the command line, 3) implementing the built-in commands, and 4) running other commands. I recommend you implement them in that order.

Implementing the Path

I recommend you create a class that uses `getenv()` to get the value of the path, and then splits it up into an array/list of directories. Then, also implement a method `find(cmd)` that searches the path directories for the given command, and returns the directory where it is found, or the empty string if it isn't found. Write unit tests to make sure your code is correct.

I found that `getenv()` and `scandir()`, among others, were useful/necessary to implement this class.

Implementing the Command Line Parser

Create a class whose constructor takes a string that will be what the user types in, and splits it into the separate words -- the command and the arguments to the command. It should check the command line for the ampersand, etc. If it finds errors, it should either print an error or raise an exception.

If an ampersand is given (correctly) the class should store this information in an instance variable (e.g., `_runInBackground`)

You should, of course, write unit tests to verify that your code is correct.

Implementing the Built-in Commands

At this point, you should start implementing `main()`. You'll want to use `getcwd()` to get the process's current working directory so you can print out the prompt. The program is a `while(True)` loop that exits when the user types `exit`.

Read the command line using your CommandLine class.

Now, implement the built-in commands. See the table above. You might/should implement the `cd` command in a separate function.

Note that you do not have to support running any built-in commands in the background.

Implementing the Non-Built-in Commands

To execute a non-built-in command, your code needs to:

- check if the location of the command was given with a full pathname. If so, check if the program exists and is executable. If it isn't, print an error.
- otherwise, find the location of the command in the path. If not found, print an error. If it is found, build up a fully-specified string indicating where the executable is located. E.g., if the user types `ps` and it is found in `/bin`, then build up the string `/bin/ps`.
- Use `fork()` and `execvp()` to run the command in a child process. You should study both functions carefully as they both have some quirks to them. Cite your sources if you use code from a website!
- Implement the code using `waitpid()` to support running a command in the background.

Inspect the example run above: you'll see that when a command is executed, the shell prints out

```
[Child pid: <pid>]
```

And, when the child process completes, the shell prints out

```
[<pid> -> <return value>]
```

The `<return value>` is the value the child process returned when it exited. If the child is run in the background, then this second informational line is not printed, as the parent cannot get the return value from the child.

Note: your shell program should allow the user to type in empty lines: it should just reprint the prompt each time.

Implementation Suggestions

You may implement this project in any language you want. I have implemented it in C, C++, and Python and I highly recommend you use Python.

I highly recommend you use unit tests.

You should read the UNIX manual pages (section 2), or the python library information, for the details of the various system calls listed above.

Spend some time thinking about all the weird stuff the user could try. Make sure you handle these cases, or document what cases you decide not to support. Your grader will be trying many weird things to see if your code handles them correctly.

Feel free to discuss the project, the Unix manual pages, and the system calls with me or your classmates. You are *not* to look at anyone else's source code.

Your program should be fully documented, with an opening (header) comment that gives all of the usual information (who, what, where, when, and why), descriptive identifiers, judicious use of white space, and in-line comments explaining any 'tricky' parts. **Write *hospitable* code** -- code that someone else is comfortable reading. Indent perfectly. Beauty counts!

Make your code modularized: make classes when possible. Make nice short functions.

Turn in

Submit your code by pushing your code to your repo. The grader will be able to pull all your code.

Grading Rubric:

22 points total:

- 4 points: header documentation, hospitable code, relatively small functions/methods, good modularization
- 3 points: command line is parsed correctly, and errors in the command line are caught.
- 3 points: built-in commands are handled correctly.
- 10 points: running child processes is done correctly, including finding the commands in the correct directories, etc.
- 2 points: submission to the correct location. See above.

This page is maintained by Victor Norman.