Lab: building bash expanding bash

Overview

In this, our final, bash lab you will download the bash source for building your own copy of bash. But first, you will be given a working C program that you will convert to a "builtin", for expanding bash's repertoire. Then you will build bash and your new builtin. Finally, you will time the new builtin and compare it to other approaches.

Step 1 - the bash source

Download a version of bash - preferably the latest - from http://mirrors.ocf.berkeley.edu/gnu/bash/ Right now the most recent version is bash-4.4.18 so get that tar.gz file and the un-tar it:

```
tar xzf bash-4.4.18.tar.gz
```

That should leave you with a `bash-4.4` directory.

Step 2 - the new command

In `cs.faculty.usna.edu/~albing/dtag.c` you will find the source for a simple C program whose purpose is to remove <tags> from text on stdin, writing the results to stdout. Get a copy of that program (but don't confuse it with the bash source; give yourself a working directory separate from the bash source). Compile `dtag.c`, convincing yourself that it works as is.

Step 3 - conversion

Shell builtins make use of a feature of the operating system called "dynamic loading" that lets you add code to a running executable. The shell command that requests such a loading is "`enable`" and is used to enable (i.e., load and make active) builtin commands that users can write themselves. To do so, though, you have to follow a strict interface.

In converting `dtag.c` to a bash builtin command, we use a slightly different name for the builtin to help avoid confusion between the two. We will call the builtin "`detag`" (adding the "e"). Copy `dtag.c` to `detag.c` and work with that version for creating the builtin. Put that version in the bash source tree under `examples/loadables` along with the other builtins. There is one particular file there: `template.c` that you can use as the basis for your new builtin. Combine a copy of that with detag.c (but don't destroy the original version of template.c!)

Looking at `template.c` you will notice that the code for a builtin has three (3) distinct sections: the main functionality, the help message text, and a C `struct` for interfacing to bash.

The easiest of these is the help message. In your source file declare, at the outer-most lexical level, an array of characters whose name is the name of your builtin with "`_doc`" added to it. Each line of the array is printed out when a user types the help command for your builtin. To see a working example, try "help enable" (we'll be using the enable command later). Keep you lines to no more than about 70 characters. Use multiple lines to explain your builtin.

The main functionality for the program is "`main`" in the original code. Give that function a new name and it will be the code that provides the functionality for your builtin.

Finally, using the "`struct builtin`" which you see as the last few lines of the template, put a similar `struct` in your `detag.c` file and modify the values accordingly.

Last look

Take one more careful read through your builtin and make sure you've replaced "template" with "detag" in all the right places. Compare your builtin to other builtins and see if you're doing enough of the required pieces.

Build bash

As with much open source code the first step is the configure command to help figure out what features your system has which can be used in the upcoming compile steps. Follow that with a make command. You'll do these two steps in the `bash-4.4` directory:

```
./configure
make
```

This has only built the standard bash command but not your builtin. Go to that directory (`examples/loadables`) and edit the Makefile in that directory. Use `unlink` or `print` as a model, and create a similar makefile target for your `detag` command. Now from within that directory:

```
make print   # just as an example, for one that works
make detag   # this is yours; does it look the same?
```

If all goes well you should go back up to the `bash-4.4` directory, run your newly built copy of bash, and the load your new detag builtin:

```
cd ../..
./bash
cd examples/loadables
enable -f ./detag  detag
```

Now you can test it out:

```
echo "This <em>is</em>a <strong>test</strong>" | detag
```

and see what you get.

Timing

Do some simple timing experiments. Run `detag` over a larger file like: `courses.cs.usna.edu/~albing/mega.txt` which you can do by runing the detag command and redirecting input from `mega.txt`. How long does it take? (remember the `time` prefix on a bash command?!)

Now compare that to some other approach - people often think of sed. Try it as a stand-alone sed command, or even as a script where you read a line a edit it (perhaps with sed). How do the times compare? Speculate on what might explain the difference. Put your measurements and your speculation in a README file and submit it along with your source file: `detag.c` to the submit system.