ACRC How-To: Customising Your Linux Environment with Environment Variables

Introduction

The *shell* is an interpreter that provides the command line interface to a Linux machine. We can type commands—with suitably arcane names such as `ls` and `top`—which are understood, and in turn executed, by the shell. Several varieties of shell exist. The most popular including `bash` (Bourne-again shell), `csh` (C shell) and `ksh` (Korn shell). Features common to all these shells include the use variables, control structures (if's and loops), filename wildcards, command pipelines etc. Commands can be strung together in text files to create more complex *shell scripts*. In this short document, we will focus on assigning values to variables in order to customise your environment. Often for the purposes of finding and using applications and executable programs. In the following examples, we will also focus on the most commonly used shell, `bash`.

Environment Variables

The `env` command (with no arguments) will report all the variables that are currently set in your environment. There are often quite a few! To simplify matters, let's just consider the variable `HOME`. To print the value of just this variable to the screen, type:

```
% echo $HOME
```

In this case, I get the value:

```
/home/gethin
```

So we see that HOME is set, by default, to the location of my home directory.

Three other environment variables, key to customising our environment for executables, are:

- **PATH**: Specifies the directories that are searched for *executable files.*
- **LD_LIBRARY_PATH**: Specifies the directories that are searched at run time for *dynamically-linked libraries* (shared-objects).
- **MANPATH**: Specifies the directories that are searched for *manual pages*.

They all take the form of colon separated lists of directories. For example, on bluecrystal phase1, I see:

```
% echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:\
/cm/shared/apps/torque/2.5.5/bin:/cm/shared/apps/torque/2.5.5/sbin:\
/cm/shared/apps/maui/3.3.1/bin:/cm/shared/apps/maui/3.3.1/sbin
```

We can see the search in action using the `which` command:

```
% which top
/usr/bin/top
```

The directories `/usr/kerberos/bin`, `/usr/local/bin` and `/bin` are all searched to see if they contain an executable program called `top`. Once a match is found in `/usr/bin`, the search is halted. It should be noted that if another executable also called `top` were located in a directory

further down the search path, it will not be found and hence will not be used.

We can specify additional directories to be searched by augmenting the list, either by appending or prefixing to the list. Prefixing ensures that any executables contained therein will be found first. Appending ensures that any previously provided versions will be preferred.

Examples

For our first example, let's append a directory in our own user file space to the search path using `bash`'s `export` command (this command ensures that the value of the given variable is exported to the environment of all subsequent commands including, for example, when one shell invokes another):

```
% export PATH=$PATH:$HOME/bin
```

Two things to note in this example are that we set the value of PATH to itself plus the additional directory. Also we can make reference to other variables in the assignment, in this case HOME.

Alternatively, to prefix to the list we could type:

```
% export PATH=$HOME/bin:$PATH
```

If an executable cannot find a dynamically-linked library at run-time, we will need to augment our LD_LIBRARY_PATH search list. Here is an example of appending a centrally located directory when using `csh`, which has a different syntax:

```
> setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:/opt/some-app/lib
```

The above assignments are valid only for a single session. If you logout and log back in again, your customisations will be lost. If you would like your changes to be present for every session on a given system, you can place commands such as those above in your *shell start-up files*.

In order to discuss start-up files, we must make a distinction between *login* shells and *normal* or *interactive non-login* shells. When we login to a machine, perhaps via SSH, a login shell is spawned. If we were to launch another shell from the login shell—by typing `csh`, or `xterm`, for example—it would be a normal, non-login shell. The reason why this is important is because different start-up files exist for both login and non-login shells. This provides sophisticated users with the ability to create nuanced shell customisations, but can also be a source of confusion for new users.

Below is a table listing the names of shell start-up files together with the shells which use them:

filename	Read and executed by:
`~/.profile`	`bash` and `ksh` login shells.
`~/.bash_profile`	`bash` login shells.
`~/.bashrc`	`bash` non-login shells.
`~/.kshrc`	`ksh` non-login shells.
`~/.bash_logout`	`bash` login shells.
`~/.login`	`csh` (and `tcsh`) login shells.
`~/.csh`	`csh` (and `tcsh`) non-login shells.
`~/.logout`	`csh` (and `tcsh`) login shells.

You can manually request that the contents of a file be read and executed using the `source`

command.

At this point, you have the choice of maintaining two start-up files, `~/.profile` and `~/.bashrc` for example, or setting up the start-up file for login shells to also source your start-up file for non-login shells. In the example below, we choose the latter approach for the `bash` shell.

Example `.bash_profile`:

```
# .bash_profile
# evaluated by login shells

# source start-up file for on-login shells:
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

Example `.bashrc`:

```
# .bashrc

alias ll='ls -l'

export PATH=$HOME/bin:$PATH
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/some-app/lib
export MANPATH=$MANPATH:/opt/some-app/man
```

In the above `.bashrc` file, we see that in addition to augmenting some search paths, we have also created a shell alias. The `alias` command allows us to create 'shortcut' commands of our own. In this case we will receive a long listing when we type ll at the prompt.

A Better Way: The Environment Modules Approach

The above approach is sometimes the only available method for shell customisation. However, it does not provide an easy way to *remove* directories from a search path once they have been added. This can lead to wasted time logging out and back into systems.

A better approach is offered by the Environment Modules package (http://modules.sourceforge.net/), which provides for the dynamic modification of a user's environment via *modulefiles*.

Each modulefile contains the information needed to configure the shell for an application. Once the Modules package is initialized, the environment can be modified on a per-module basis using the module command which interprets modulefiles. Typically modulefiles instruct the module command to alter or set shell environment variables such as PATH, MANPATH, etc. Modulefiles may be shared by many users on a system and users may have their own collection to supplement or replace the shared set.

Modules can be **loaded** and **unloaded** dynamically and atomically, in an clean fashion. All popular shells are supported, including *bash*, *ksh*, *zsh*, *sh*, *csh*, *tcsh*, as well as some scripting languages such as *perl* and *python*.

Modules are useful in managing different versions of applications. Modules can also be bundled into metamodules that will load an entire suite of different applications.

Examples

If a system uses environment modules, we would first like to discover if an modules are currently loaded. On bluecrystal phase1, for example, we might see:

```
% module list
Currently Loaded Modulefiles:
   1) torque/2.5.5           2) maui/3.3.1
```

Next, we might ask to see what other modules are available to be loaded. On bluecrystal, this typically results in a long list. Below is a representative sample:

```
% module avail
apps/abaqus-6.10                 languages/R-2.15.1
apps/autodock-4.2.3              languages/gcc-4.8.0
apps/comsol4.1                   languages/java-jdk-1.7.0
apps/comsol4.3                   languages/mono-3.0.1
apps/fsl                         languages/perl-5.14
apps/gmsh-2.5.0                  languages/pgi-12.8
apps/gnuplot-4.4.4               languages/python-2.7.2
apps/grace-5.1.20                libraries/gcc-4.4.6-mpi-fftw-2.15.1
apps/ls-dyna971-r6.1.0           libraries/gcc-4.4.6-serial-fftw-2.15.1
apps/matlab-R2011a               libraries/gcc-4.6.2-atlas-3.8.4
apps/mr-bayes-3.1.2-Dirichlet    libraries/gcc-netcdf-4.0
apps/neuron-7.2                  libraries/intel-12-slatec
apps/paml-4.5                    libraries/intel-12.1.2-atlas-3.8.4
apps/panoply-3.1.5               libraries/intel-12.1.2-hdf5
apps/rocb-1.1.1                  libraries/intel-12.1.2-netcdf-4.1.1
apps/stata12                     libraries/nag-fortran-intel-12-64-bit
apps/trinity-2012-01-25          libraries/nag-fortran-pgi-64-bit
base/infiniband-gcc              maui/3.3.1
base/infiniband-intel            open64/4.2.5
base/infiniband-pgi              openmpi/gcc/64/1.4.4
base/openmp-gcc                  openmpi/intel/64/1.4.4
base/openmp-pgi                  openmpi/pgi/64/1.4.4
blas/open64/64/1                 pgi/64/11.10
blas/pgi/64/1                    slurm/2.2.4
cmgui/5.2                        tools/cmake-2.8.1
default-environment              tools/git-1.7.9
gcc/4.4.6                        tools/subversion-1.7.3
gpu/cuda-toolkit-4.2.9           tools/tau-2.21.1-openmp
intel/tbb/64/4.0/2011_sp1.8.273  tools/valgrind-3.7.0
```

Having identified a desirable module, we can load it:

```
% module load base/infiniband-gcc
```

This is a metamodule that provides access to not only the `gcc` compiler, but also MPI commands such as `mpicc`:

```
% which mpicc
/cm/shared/apps/openmpi/gcc/64/1.4.4/bin/mpicc
```

Now we'll switch to a different version of the module

```
% module switch base/infiniband-gcc base/infiniband-intel
% which mpicc
/cm/shared/apps/openmpi/intel/64/1.4.4/bin/mpicc
```

Or unload the module altogether

```
% module unload base/infiniband-intel
% which mpicc
mpicc: Command not found.
```

Adding Module Commands to Your Shell Start-Up Files

Just as we placed commands to set environment variables directly in our shell start-up files, we can also include module commands so that they will be executed automatically when a shell is created. For example, we always load the `base/infiniband-gcc` in our `bash` shell using:

```
# .bashrc

alias ll='ls -l'

module load base/infiniband-gcc
```

Creating Your Own Module Files

Your use of modulefiles need not be limited to those provided by your system administrator as you can create your own modules, which you might use to provide convenient access to applications that you have stored in your own userspace, or indeed to customise your use of centrally provided applications.

Installing applications for use with modules

Let's suppose that you have the computer algebra application PARI/GP installed under your home directory. To keep things manageable, it is best to use a separate sub-directory for each application. Installing PARI/GP into `$HOME/moduleapps/pari` gives the following directories:

`$HOME/moduleapps/pari/bin`	commands to run
`$HOME/moduleapps/pari/lib`	libraries
`$HOME/moduleapps/pari/include`	header files for using the PARI libraries in C programs
`$HOME/moduleapps/pari/share/man`	man pages
`$HOME/moduleapps/pari/share/pari`	various data files for PARI

Adopting the above approach, other applications would be installed in their own directories alongside this, e.g.:

```
$HOME/moduleapps/app1
$HOME/moduleapps/app2
```

With their own separate bin, lib and other sub-directories. In this way, the application installations

do not overlap with one another at all.

Handling this manually you'd add the `bin` directory to the `PATH` environment variable, the `lib` directory to your LD_LIBRARY_PATH and so on, as shown earlier. Instead, we will create a module file so that we are able to easily grow and shrink our path variables for a given application. Using this approach will also make it easier for us to keep track environment customisations when we desire access to several applications.

Writing personalised modules

The command:

```
% module load use.own
```

will add the `$HOME/privatemodules` directory to the module search path. This allows us to add module files to this directory and have them show up when you use the `module avail` command.

Next, you need to write a module file. A simple example is given below for the example PARI/GP installation, given earlier. Lines which begin with a hash character are comments.

```
#%Module 1.0
#
# PARI/GP and associated tools and libraries
#
module-whatis “Adds PARI/GP and associated tools and \
libraries to your environment variables”
#
# The "set root" line just creates a variable to make the
# subsequent paths shorter:
set root ~/moduleapps/pari
prepend-path PATH            $root/bin
prepend-path LD_LIBRARY_PATH $root/lib
prepend-path MANPATH         $root/share/man
#
# Arbitrary environment variables can be set as follows:
setenv       SOME_VARIABLE   "PARI required variable value"
```

Save this file as `~/privatemodules/pari` and it will be available to `module load` commands.

Note how the `PATH`, `LD_LIBRARY_PATH` and `MANPATH` variables are prepended to for this particular application directory; hence the requirement to separate applications and allow them to be individually loaded and unloaded. Arbitrary environment variables which some applications require, say for paths the licence servers, can also be set in these module files.

If you want to have multiple different versions of an application selectable as modules, you can simply install the versions into different directories and create separate modules for them. For example, you can install gcc-4.6.3 and gcc-4.7.1 into different directories and then create `gcc-4.6.3` and `gcc-4.7.1` modulefiles, allowing you to use `module load gcc-<version>` to choose between them.

Loading your new module

Once you have applications installed in this way and have used `module load use.own` to add the `privatemodules` directory to the module search path, the `module avail` command will show these additions:

```
% module avail

[other modules directories listed...]

---------------------------------------
/home/username/privatemodules
---------------------------------------
pari
```

and you can use `module load pari` to appropriately customise your environment.

You can load the `use.own` module and any application modules you wish to use in a single line in your shell start-up file. For example:

```
# .bashrc
#
# [other bashrc entries, aliases etc]

module load use.own pari
```

Appendix

Two other path variables which you might find useful are described below.

LIBRARY_PATH

`LIBRARY_PATH` is similar to `LD_LIBRARY_PATH` but is used at compile-time to allow the compiler and linker to find libraries without the need to explicitly specify the full path on the build command-line.

For example, the command:

```
% gcc a.o b.o -L/some/path/to/a/lib/dir -lfoo
```

can be simplified by setting `LIBRARY_PATH` in a module:

```
  prepend-path LIBRARY_PATH /some/path/to/a/lib/dir
```

After which, the -L option can be omitted:

```
% gcc a.o b.o -lfoo
```

CPATH

CPATH is almost identical to LIBRARY_PATH but is for use with C and Fortran90 compilers to find header and module files. To illustrate it's use, the command:

```
% gcc -c file.c -o file.o -I/some/path/to/an/include/dir
```

can again be simplified, providing CPATH is set in a module:

```
  prepend-path CPATH /some/path/to/an/include/dir
```

Where, now we no longer need to pass the -I option:

```
% gcc -c file.c -o file.o
```

Both CPATH and LIBRARY_PATH make it simpler to switch between different versions of libraries, as the Makefiles (or equivalent) for your code to compile will not need modification. Simply load different modules and the path settings will be updated to find the alternative version.