

From SIEpedia

SIEminars: EnvironmentVariables

Many Unix/Linux programs, including the shell, need information about your account and what you are doing in order to work correctly and efficiently.

For instance, the shell may need to know what kind of terminal you are using, or what your home directory is, or in what directory it must look for commands and executables.

So, UNIX uses **environment variables** (EV for short) to store information that you'd rather not worry about, either because it's something specific to your platform or shell, or because it is set by your system manager.

Environment variables are managed by the shell, so different shells may have different sets of EVs, and they can be defined in a different way. (This often causes confusion among users of the tcsh shell when they try to set the EVs after logging in as root in a bash shell).

This presentation applies to a tcsh shell in a desktop Linux PC.

How to list, define and modify Environment Variables

A list of all defined EVs can be obtained by using the command `printenv`:

```
printenv
```

To ask about a particular variable:

```
printenv HOME
```

Alternatively (note the dollar sign):

```
echo $HOME
```

To set or redefine a variable, the command is **setenv NAME value**. For instance:

```
setenv EDITOR /usr/local/bin/emacs
setenv PRINTER lw3
```

To have the shell forget that an environment variable ever existed, use the **unsetenv** command, for instance:

```
unsetenv TINYTIM
```

By convention, the names of environmental variables are all uppercase, though there's nothing to enforce this convention.

Digression: The () Subshell Operator

A quick digression on the subshell operator. If we wish to run some commands in a subshell, so as not to affect the primary shell, the *() subshell operator* is a very useful tool.

For instance, if we wish to know what time it is in Vladivostok, we can accomplish it by changing accordingly the TimeZone TZ variable, and then executing the date command:

```
(setenv TZ Asia/Vladivostok ; date)
```

By doing it in a subshell, we do not need to remember what value TZ had before, and change it back after running the date command.

This trick is also useful, for example, to run a command in a different directory without changing the current work directory:

```
(cd $HOME ; latex mypaper ; dvips mypaper -o)
```

You can use the subshell operator to test definitions or modifications of variables before committing to them.

Parent-Child Relationships

Each subprocess, or a new subshell, will get its own copy of its parent's environment variables. For instance, if you write and execute a shell script, it will "see", or "inherit", all the environment variables present in the parent shell.

However, the opposite is not true: a UNIX process can not change its parent's environment; any changes it keeps to itself. So, if you write a shell script where you define new EVs, and start it, when it's finished there is no trace of them.

An easy way to check it is to use a subshell:

```
( printenv )
( setenv OFFICE 2104 ; printenv OFFICE ) ; printenv OFFICE
```

variable OFFICE is undefined in the parent shell.

Some of the Most Important Environment Variables

Here we'll have a look at those Environment Variables which are specially important, and sometimes need some modifications or changes by the user (for instance after installing a new software package).

PATH

It contains the *command search path*, that is a list of directories in which the shell looks to find commands. In particular, whenever a command is typed, the shell will go through the list of directories, in the order they appear, until it finds it.

```
printenv PATH
```

This is not completely true, though. Actually the shell doesn't search directly the path, but uses a hash table to find the command quickly. That's why, if a new executable is added in any of the directories in PATH, you need to type **rehash** to have the shell rebuild the hash table.

As the order in \$PATH is important, one should pay attention to whether prepending or appending a new directory to this variable.

An empty entry in PATH (: as first or last character, or :: in the middle) means "the current directory". So, if you can't launch an executable in the current directory by just typing its name (you get an error message like "Command not found"), add the current directory to the PATH.

```
setenv PATH ${PATH}:
```

Generally it is not a problem to have directories appear twice or more times in the PATH, unless the variable becomes too long (the limit is typically 1024 bytes).

To add a directory to PATH, use the following syntax:

```
setenv PATH ${PATH}:/path/for/new/directory
-or-
setenv PATH /path/for/new/directory:${PATH}
```

Be very careful with the syntax, otherwise you can delete or set the variable in a incorrect or inconsistent way. For example:

```
(setenv PATH ; ls ; who)
PATH is set to null by mistake, and nothing works anymore.
```

LD_LIBRARY_PATH

This is a list of directoris where the shell looks for *shared object libraries*. If you install a new package, with some new shared objects, you need either copy or link those files into a directory already included in LD_LIBRARY_PATH, or add the new directory to LD_LIBRARY_PATH.

```
printenv LD_LIBRARY_PATH
```

If, when you launch some executable, you obtains error messages like "error while loading shared libraries: libtermcap.so.2", it means that either the missing library is not installed, or it's installed in a directory not included in LD_LIBRARY_PATH.

Note that Linux has already a default set of directories where to look for libraries.

To add a directory to LD_LIBRARY_PATH, follow the same prescriptions given for PATH, for instance:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:/home/ncaon/lib_Linux
```

DISPLAY

It is used by the X Window System to identify the display server that will be used for input and output. It must be defined if we connect remotely to another machine. In principle the **ssh** command should take care of defining it correctly, but sometimes it does not. So you may need to set it manually, for instance:

```
setenv DISPLAY cherne:0.0
```

LC_ALL, LANG

These variables define the language used by your shell. Their default value is "es_ES", that is spanish.

Please take into account that some programs or scripts may fail or behave strangely if run in a spanish-speaking shell. Moreover, somebody may find the error messages printed in spanish somewhat funny (for instance, *Violación de segmentos*: is this some sort of sexual crime that a polygon might commit?) and prefers to interact with an english-speaking shell.

If this is your case, you can redefine the above variables to, e.g., *American English*:

```
setenv LC_ALL en_US
setenv LANG en_US
```

Ohter languages are also available, and you can play around with them. For instance see how "command not found" is translated:

```
(setenv LC_ALL en_US ; setenv LANG en_US ; aaaaa)
(setenv LC_ALL es_ES ; setenv LANG es_ES ; aaaaa)
(setenv LC_ALL fi_FI ; setenv LANG fi_FI ; aaaaa)
```

```
(setenv LC_ALL de_DE ; setenv LANG de_DE ; aaaaa)
(setenv LC_ALL fr_FR ; setenv LANG fr_FR ; aaaaa)
```

By who and where are EVs defined?

Broadly speaking, we can divide the Environment Variables into three groups:

- Variables set (and modified automatically) by the Operating System and by the shell. This is usually done by startup files such as `/etc/csh.cshrc` and `/etc/csh.login`, which in turn "source" other files. Such variables are, for instance:
 - ◆ HOME;
 - ◆ USER;
 - ◆ HOST;
 - ◆ PWD, contains the absolute pathname of the current directory, and it's reset automatically by the `cd` command;
 - ◆ SHELL, the absolute pathname of your login shell;
 - ◆ TERM, which identifies the type of terminal you are using;
 - ◆ PATH and (perhaps) LD_LIBRARY_PATH, but only including the basic directories;
- Variables set or modified by the System Administrators (see later on) to best configure your environment and allow locally installed programs to work:
 - ◆ PATH and LD_LIBRARY_PATH are expanded by adding the relevant directories for programs such as Latex, Intel Fortran compiler, Java, etc.;
 - ◆ LANG, LANGVAR and LC_ALL set the language you are working in (I think spanish id the default);
 - ◆ PGPLOT_DIR, PGPLOT_FONT, required by pgplot
 - ◆ IRAFARCH, which tells Iraf the architecture of the machine (for instance *redhat*, or *ssun*)
- Variables set or modified (with care) by the users themselves, in their startup files (see below).

Shell variables

Shell variables are pigeonholes that store information for you or the shell to use. How are they different from environment variables? If you start a new subshell or a program, it inherits all of its parent's environment variables, but does not inherit any shell variables. In a sense, environment variables are "global", shell variables are "local".

By convention, shell variables are all lowercase.

To define or modify a shell variable, use the **set variable=value** command:

```
set history=1000
```

This changes how many previous commands the shell will remember.

To delete a variable, use the **unset** command

```
unset history
```

Note the undefining a variable is not the same as defining it to a null value, as some programs look at variables to see whether they exist or not, regardless of their value.

To list all shell variables, use the **set** command:

```
set
```

LC_ALL, LANG

Among the most important shell variables, we can find:

- **path**, which is identical to PATH (modifying one automatically changes the other)
- **autolist**, which enables completion of file or command names by using the TAB key (after typing each of these commands, check whether the TAB key expands or not the list of file or command names matching what you typed):

```
set autolist
unset autolist
```
- **nobeep**: if you get annoyed by your terminal beeping mercilessly every time you do something wrong, do:

```
set nobeep
```
- **prompt**: set your prompt

```
set prompt = "%t$"
set prompt = "%{\033]0;%n@~\007%}%n@%m> "
```

The first command will set the prompt to the current time; the second, somewhat more cryptic (it uses some shell escape commands), will set the prompt to "user@machine > ", and will repeat the prompt, together with the current directory, in the xterm title bar (seeing is believing).
- **correct**: if set to *cmd*, the shell will try to perform spelling correction on a misspelled command.

```
set correct=cmd
```

(try misspelling some commands, try again after undefining the variable).

There are further shell variables that may be of some interest. Please see the tcsh man page.

User's Startup files

Sad but true stories ...

Excerpts from some CAU tickets:

Q.: All of a sudden I can't access my home

A.: Files .cshrc and .login are missing. Standards .cshrc and .login files are copied into the user's home.

Q.: I can't login into my account anymore! I only did a little change to the PATH variable in my .cshrc file...

A.: PATH was changed using an incorrect syntax, making this variable useless.

Q.: Why are my print jobs sent to printer lwA, after I asked the CAU to configure my machine with printer lwB as the default one?

A.: The printer was somehow redefined in the user's .login file.

Q.: I can't run Latex in my machine!

A.: The user had commented out, in his .cshrc file, the line:

```
# source /usr/glob/user/.cshrc
```

What are the basic startup files?

The tcsh shell has basically three shell setup files available:

- **.tcshrc** or **.cshrc** : It is read anytime a shell starts, including shell escapes and shell scripts (unless started using the *-f* flag). This is the place to put commands that should run every time you start a shell (for instance open a new terminal window). Here go the prompt and aliases.

If the tcsh shell can't find this file, it will look for file **.cshrc**. If you have either file `.tcshrc` or file `.cshrc` in your home, you're OK; having both is certainly a source of confusion (more for the user than for the system), and should be avoided. (Hereinafter we will refer only to `.cshrc`)

Before reading the user's `.cshrc` file, the shell reads `/etc/csh.cshrc`

- **.login**: It is read when you start a login shell. Here, in principle, you should put environment variables, terminal-related commands, and other commands you want to run everytime you log in. Note that `.cshrc` is read before `.login`.

Before reading the user's `.login` file, the shell reads `/etc/csh.login`

- **.logout**: It is read when you end a login shell. This is rarely used. It may include some "goodbye" message, or the command **clear** to clear the terminal, or a command to clean up temporary files. Before this file, the shell reads `/etc/csh.logout` (may depend on specific distribution).

Login and non-login shells

While the distinction between login and non-login shells was clear in the old days, now it's somewhat blurred. For instance Linux has interactive shells started by the windows system, remote shells, and other shells that, depending on how they are invoked, might or might not be login shells. So it is not always clear when a new shell or application reads or not the `.login` file.

Empirically, I've found the following (by having `.cshrc` and `.login` print a message when they are read):

- Konsole, XTerm, XGterm: all are non-login shells (so they do not read `.login`), unless they are started with the flag `"-ls"`.
- ssh: opens a login shell.
- tcsh (new subshell): reads only `.cshrc`
- () subshell: reads neither `.cshrc` nor `.login`.

How can this be handled? A practical solution is to put everything in the `.cshrc` file, except perhaps some commands such as printing the "message of the day" or a "fortune". On the other hand, since `.cshrc` is read by every shell and sub-shell (unless instructed otherwise), it is good practice not to overload it with unnecessary stuff.

Standard .cshrc and .login files: do's and dont's

Every time a new account is created, it comes with default `.cshrc` and `.login` files, set up by the system administrators.

They should be modified with much care. For instance, do not delete or comment out the line, in the `.cshrc` file: `source /usr/glob/user/.cshrc`

This file in turn sources several other configuration files, which all together build and define your environment. If you remove this line, your environment will be incomplete and many things won't work!

Also, in `.login`, heed the warning and do not put anything below the last line, where the global `.login` file is sourced.

If you are doing some file input or output in a setup file (such as reading or sourcing from a file, or writing some date in a log file), please use absolute pathnames, not relative ones. For example:

```
source .aliases # Wrong!
echo "Logged in at `date`" >> login.log # Wrong!
```

What are the basic startup files?

```
source ~/.aliases # OK!
echo "Logged in at `date`" >> ~/login.log # OK!
```

The first two lines won't work if the (sub)shell is started from somewhere else than the home directory.

Keep the .cshrc and .login clear and neat

A lot of stuff can go inside a .cshrc files: new or redefined environment and shell variables, aliases, prompts, perhaps some conditional instructions depending on which machine or platform you are using, etc.

So, instead of ending up with a longish .cshrc files including thousands of instructions, you can organize them in separate files. A .cshrc file might look like this:

[[Get Code](#)]

```
# Default instruction placed by the System Administrator
```

```
source /usr/glob/user/.cshrc
```

```
# Modify path to append my personal bin directory:
```

```
setenv PATH ${PATH}~/mybin
```

```
# Define environment variables needed by some personal packages
```

```
source ~/.set_environment
```

```
# Load aliases:
```

```
source ~/.aliases
```

```
# Set shell variables:
```

```
source ~/.shell_variables
```

```
# Platform dependent stuff:
```

```
if ( $OSTYPE == "Linux" ) then
```

```
    source ~/.linux_specific
```

```
endif
```

```
# Host dependent stuff (for instance a burro, or a machine with some specific configuration):
```

```
if ( $HOST == "esel" ) then
```

```
    source ~/.esel_specific
```

```
endif
```

Retrieved from <http://www.iac.es/sieinvens/siepedia/pmwiki.php?n=SIeminars.EnvironmentVariables>

Page last modified on March 17, 2009, at 05:19 PM