

Using Linux Environment Variables

IN THIS CHAPTER

Looking at environment variables

Creating your own local variables

Removing variables

Exploring default shell environment variables

Setting the PATH environment variable

Locating environment files

Using variable arrays

Linux environment variables help define your Linux shell experience. Many programs and scripts use environment variables to obtain system information and store temporary data and configuration information. Environment variables are set in lots of places on the Linux system, and you should know where these places are.

This chapter walks you through the world of Linux environment variables, showing where they are, how to use them, and even how to create your own. The chapter finishes off with how to use variable arrays.

Exploring Environment Variables

The bash shell uses a feature called *environment variables* to store information about the shell session and the working environment (thus the name environment variables). This feature also allows you to store data in memory that can be easily accessed by any program or script running from the shell. It is a handy way to store needed persistent data.

There are two environment variable types in the bash shell:

- Global variables
- Local variables

Linux® Command Line and Shell Scripting Bible, Third Edition. Richard Blum, Christine Bresnahan.
© 2015 by John Wiley & Sons, Inc. Published 2015 by John Wiley & Sons Inc.

This section describes each type of environment variable and shows how to see and use them.

NOTE

Even though the bash shell uses specific environment variables that are consistent, different Linux distributions often add their own environment variables. The environment variable examples you see in this chapter may differ slightly from what's available on your specific distribution. If you run into an environment variable not covered here, check your Linux distribution's documentation.

Looking at global environment variables

Global environment variables are visible from the shell session and from any spawned child subshells. Local variables are available only in the shell that creates them. This makes global environment variables useful in applications that create child subshells, which require parent shell information.

The Linux system sets several global environment variables when you start your bash session. (For more details about what variables are started at that time, see the “Locating System Environment Variables” section later in this chapter.) The system environment variables almost always use all capital letters to differentiate them from normal user environment variables.

To view global environment variables, use the `env` or the `printenv` command:

```
$ printenv
HOSTNAME=server01.class.edu
SELINUX_ROLE_REQUESTED=
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
[...]
HOME=/home/Christine
LOGNAME=Christine
[...]
G_BROKEN_FILENAMES=1
_=/usr/bin/printenv
```

So many global environment variables get set for the bash shell that the display had to be snipped. Not only are many set during the login process, but how you log in can affect which ones are set as well.

To display an individual environment variable's value, you can use the `printenv` command, but not the `env` command:

```
$ printenv HOME
/home/Christine
```

```
$
$ env HOME
env: HOME: No such file or directory
$
```

You can also use the `echo` command to display a variable's value. When referencing an environment variable in this case, you must place a dollar sign (\$) before the environment variable name:

```
$ echo $HOME
/home/Christine
$
```

Using the dollar sign along with the variable name does more than just display its current definition when used with the `echo` command. The dollar sign before a variable name allows the variable to be passed as a command parameter:

```
$ ls $HOME
Desktop    Downloads  Music      Public     test.sh
Documents  junk.dat   Pictures   Templates  Videos
$
$ ls /home/Christine
Desktop    Downloads  Music      Public     test.sh
Documents  junk.dat   Pictures   Templates  Videos
$
```

As mentioned earlier, global environment variables are also available to any process's subshells:

```
$ bash
$
$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
501          2017   2016  0 16:00 pts/0    00:00:00 -bash
501          2082   2017  0 16:08 pts/0    00:00:00 bash
501          2095   2082  0 16:08 pts/0    00:00:00 ps -f
$
$ echo $HOME
/home/Christine
$
$ exit
exit
$
```

In this example, after spawning a subshell using the `bash` command, the `HOME` environment variable's current value is shown. It is set to the exact same value, `/home/Christine`, as it was in the parent shell.

Looking at local environment variables

Local environment variables, as their name implies, can be seen only in the local process in which they are defined. Even though they are local, they are just as important as global environment variables. In fact, the Linux system also defines standard local environment variables for you by default. However, you can also define your own local variables. These, as you would assume, are called user-defined local variables.

Trying to see the local variables list is a little tricky at the CLI. Unfortunately, there isn't a command that displays only these variables. The `set` command displays all variables defined for a specific process, including both local and global environment variables and user-defined variables:

```
$ set
BASH=/bin/bash
[...]
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
[...]
colors=/etc/DIR_COLORS
my_variable='Hello World'
[...]
$
```

All global environment variables displayed using the `env` or `printenv` commands appear in the `set` command's output. The additional environment variables are the local environment and user-defined variables.

NOTE

The differences between the commands `env`, `printenv`, and `set` are subtle. The `set` command displays both global and local environment variables and user-defined variables. It also sorts the display alphabetically. The `env` and `printenv` are different from `set` in that they do not sort the variables, nor do they include local environment or local user-defined variables. Used in this context, `env` and `printenv` produce duplicate listings. However, the `env` command has additional functionality that `printenv` does not have, making it the slightly more powerful command.

Setting User-Defined Variables

You can set your own variables directly from the bash shell. This section shows you how to create your own variables and reference them from an interactive shell or shell script program.

Setting local user-defined variables

After you start a bash shell (or spawn a shell script), you're allowed to create local user-defined variables that are visible within your shell process. You can assign either a numeric or a string value to an environment variable by assigning the variable to a value using the equal sign:

```
$ echo $my_variable  
  
$ my_variable=Hello  
$  
$ echo $my_variable  
Hello
```

That was simple! Now, any time you need to reference the `my_variable` user-defined variable's value, just reference it by the name `$my_variable`.

If you need to assign a string value that contains spaces, you need to use a single or double quotation mark to delineate the beginning and the end of the string:

```
$ my_variable=Hello World  
-bash: World: command not found  
$  
$ my_variable="Hello World"  
$  
$ echo $my_variable  
Hello World  
$
```

Without the quotation marks, the bash shell assumes that the next word is another command to process. Notice that for the local variable you defined, you used lowercase letters, while the system environment variables you've seen so far have all used uppercase letters.

Tip

The standard bash shell convention is for all environment variables to use uppercase letters. If you are creating a local variable for yourself and your own shell scripts, use lowercase letters. Variables are case sensitive. By keeping your user-defined local variables lowercase, you avoid the potential disaster of redefining a system environment variable.

It's extremely important that you not use spaces between the variable name, the equal sign, and the value. If you put any spaces in the assignment, the bash shell interprets the value as a separate command:

```
$ my_variable = "Hello World"  
-bash: my_variable: command not found  
$
```

After you set a local variable, it's available for use anywhere within your shell process. However, if you spawn another shell, it's not available in the child shell:

```
$ my_variable="Hello World"
$
$ bash
$
$ echo $my_variable

$ exit
exit
$
$ echo $my_variable
Hello World
$
```

In this example, a child shell was spawned. The user-defined `my_variable` was not available in the child shell. This is demonstrated by the blank line returned after the `echo $my_variable` command. After the child shell was exited and returned to the original shell, the local variable was still available.

Similarly, if you set a local variable in a child process, after you leave the child process, the local variable is no longer available:

```
$ echo $my_child_variable

$ bash
$
$ my_child_variable="Hello Little World"
$
$ echo $my_child_variable
Hello Little World
$
$ exit
exit
$
$ echo $my_child_variable

$
```

The local variable set within the child shell doesn't exist after a return to the parent shell. You can change this behavior by turning your local user-defined variable into a global environment variable.

Setting global environment variables

Global environment variables are visible from any child processes created by the parent process that sets the variable. The method used to create a global environment variable is to first create a local variable and then export it to the global environment.

This is done by using the `export` command and the variable name minus the dollar sign:

```
$ my_variable="I am Global now"
$
$ export my_variable
$
$ echo $my_variable
I am Global now
$
$ bash
$
$ echo $my_variable
I am Global now
$
$ exit
exit
$
$ echo $my_variable
I am Global now
$
```

After defining and exporting the local variable `my_variable`, a child shell was started by the `bash` command. The child shell was able to properly display the `my_variable` variable's value. The variable kept its value, because the `export` command made it a global environment variable.

Changing a global environment variable within a child shell does not affect the variable's value in the parent shell:

```
$ my_variable="I am Global now"
$ export my_variable
$
$ echo $my_variable
I am Global now
$
$ bash
$
$ echo $my_variable
I am Global now
$
$ my_variable="Null"
$
$ echo $my_variable
Null
$
$ exit
exit
$
```

```
$ echo $my_variable
I am Global now
$
```

After defining and exporting the variable `my_variable`, a subshell was started by the `bash` command. The subshell properly displayed the value of the `my_variable` global environment variable. The variable's value was then changed by the child shell. However, the variable's value was modified only within the child shell and not in the parent's shell.

A child shell cannot even use the `export` command to change the parent shell's global environment variable's value:

```
$ my_variable="I am Global now"
$ export my_variable
$
$ echo $my_variable
I am Global now
$
$ bash
$
$ echo $my_variable
I am Global now
$
$ my_variable="Null"
$
$ export my_variable
$
$ echo $my_variable
Null
$
$ exit
exit
$
$ echo $my_variable
I am Global now
$
```

Even though the child shell redefined and exported the variable `my_variable`, the parent shell's `my_variable` variable kept its original value.

Removing Environment Variables

Of course, if you can create a new environment variable, it makes sense that you can also remove an existing environment variable. You can do this with the `unset` command. When referencing the environment variable in the `unset` command, remember not to use the dollar sign:

```
$ echo $my_variable
I am Global now
```



```
$  
$ unset my_variable  
$  
$ echo $my_variable  
  
$
```

Tip

It can be confusing to remember when to use and when not to use the dollar sign with environment variables. Just remember this: If you are doing anything *with* the variable, use the dollar sign. If you are doing anything *to* the variable, don't use the dollar sign. The exception to this rule is using `printenv` to display a variable's value.

When dealing with global environment variables, things get a little tricky. If you're in a child process and unset a global environment variable, it applies only to the child process. The global environment variable is still available in the parent process:

```
$ my_variable="I am Global now"  
$  
$ export my_variable  
$  
$ echo $my_variable  
I am Global now  
$  
$ bash  
$  
$ echo $my_variable  
I am Global now  
$  
$ unset my_variable  
$  
$ echo $my_variable  
  
$ exit  
exit  
$  
$ echo $my_variable  
I am Global now  
$
```

Just as with modifying a variable, you cannot unset it in a child shell and have the variable be unset in the parent's shell.

Uncovering Default Shell Environment Variables

The bash shell uses specific environment variables by default to define the system environment. You can always count on these variables being set or available to be set on your

Linux system. Because the bash shell is a derivative of the original Unix Bourne shell, it also includes environment variables originally defined in that shell.

Table 6-1 shows the environment variables that the bash shell provides that are compatible with the original Unix Bourne shell.

TABLE 6-1 The bash Shell Bourne Variables

Variable	Description
CDPATH	A colon-separated list of directories used as a search path for the <code>cd</code> command
HOME	The current user's home directory
IFS	A list of characters that separate fields used by the shell to split text strings
MAIL	The filename for the current user's mailbox (The bash shell checks this file for new mail.)
MAILPATH	A colon-separated list of multiple filenames for the current user's mailbox (The bash shell checks each file in this list for new mail.)
OPTARG	The value of the last option argument processed by the <code>getopt</code> command
OPTIND	The index value of the last option argument processed by the <code>getopt</code> command
PATH	A colon-separated list of directories where the shell looks for commands
PS1	The primary shell command line interface prompt string
PS2	The secondary shell command line interface prompt string

Besides the default Bourne environment variables, the bash shell also provides a few variables of its own, as shown in Table 6-2.

TABLE 6-2 The bash Shell Environment Variables

Variable	Description
BASH	The full pathname to execute the current instance of the bash shell
BASH_ALIASES	An associative array of currently set aliases
BASH_ARGC	A variable array that contains the number of parameters being passed to a subroutine or shell script
BASH_ARCV	A variable array that contains the parameters being passed to a subroutine or shell script
BASH_CMDS	An associative array of locations of commands the shell has executed

BASH_COMMAND	The shell command currently being or about to be executed
BASH_ENV	When set, each bash script attempts to execute a startup file defined by this variable before running.
BASH_EXECUTION_STRING	The command(s) passed using the bash -c option
BASH_LINENO	A variable array containing the source code line number of the currently executing shell function
BASH_REMATCH	A read-only variable array containing patterns and their sub-patterns for positive matches using the regular expression comparison operator, =~
BASH_SOURCE	A variable array containing the source code filename of the currently executing shell function
BASH_SUBSHELL	The current nesting level of a subshell environment (The initial value is 0.)
BASH_VERSINFO	A variable array that contains the individual major and minor version numbers of the current instance of the bash shell
BASH_VERSION	The version number of the current instance of the bash shell
BASH_XTRACEFD	If set to a valid file descriptor (0,1,2), trace output generated from the 'set -x' debugging option can be redirected. This is often used to separate trace output into a file.
BASHOPTS	A list of bash shell options that are currently enabled
BASHPID	Process ID of the current bash process
COLUMNS	Contains the terminal width of the terminal used for the current instance of the bash shell
COMP_CWORD	An index into the variable COMP_WORDS, which contains the current cursor position
COMP_LINE	The current command line
COMP_POINT	The index of the current cursor position relative to the beginning of the current command
COMP_KEY	The final key used to invoke the current completion of a shell function
COMP_TYPE	An integer value representing the type of completion attempted that caused a completion shell function to be invoked
COMP_WORDBREAKS	The Readline library word separator characters for performing word completion
COMP_WORDS	An array variable that contains the individual words on the current command line
COMPREPLY	An array variable that contains the possible completion codes generated by a shell function

Continues

TABLE 6-2 (continued)

Variable	Description
COPROC	An array variable that holds an unnamed coprocess' I/O file descriptors
DIRSTACK	An array variable that contains the current contents of the directory stack
EMACS	Indicates the emacs shell buffer is executing and line editing is disabled, when set to 't'
ENV	When set, executes the startup file defined before a bash shell script runs (It is used only when the bash shell has been invoked in POSIX mode.)
EUID	The numeric effective user ID of the current user
FCEDIT	The default editor used by the <code>fc</code> command
FIGIGNORE	A colon-separated list of suffixes to ignore when performing file name completion
FUNCNAME	The name of the currently executing shell function
FUNCNEST	Sets the maximum allowed function nesting level, when set to a number greater than zero (If it is exceeded, the current command aborts.)
GLOBIGNORE	A colon-separated list of patterns defining the set of filenames to be ignored by file name expansion
GROUPS	A variable array containing the list of groups of which the current user is a member
histchars	Up to three characters, which control history expansion
HISTCMD	The history number of the current command
HISTCONTROL	Controls what commands are entered in the shell history list
HISTFILE	The name of the file in which to save the shell history list (<code>.bash_history</code> by default)
HISTFILESIZE	The maximum number of lines to save in the history file
HISTTIMEFORMAT	Used as a formatting string to print each command's timestamp in bash history, if set and not null
HISTIGNORE	A colon-separated list of patterns used to decide which commands are ignored for the history file
HISTSIZE	The maximum number of commands stored in the history file
HOSTFILE	Contains the name of the file that should be read when the shell needs to complete a hostname
HOSTNAME	The name of the current host
HOSTTYPE	A string describing the machine the bash shell is running on

IGNOREEOF	The number of consecutive EOF characters the shell must receive before exiting (If this value doesn't exist, the default is 1.)
INPUTRC	The name of the Readline initialization file (The default is <code>.inputrc</code> .)
LANG	The locale category for the shell
LC_ALL	Overrides the <code>LANG</code> variable, defining a locale category
LC_COLLATE	Sets the collation order used when sorting string values
LC_CTYPE	Determines the interpretation of characters used in filename expansion and pattern matching
LC_MESSAGES	Determines the locale setting used when interpreting double-quoted strings preceded by a dollar sign
LC_NUMERIC	Determines the locale setting used when formatting numbers
LINENO	The line number in a script currently executing
LINES	Defines the number of lines available on the terminal
MACHTYPE	A string defining the system type in <code>cpu-company-system</code> format
MAPFILE	An array variable that holds read-in text from the <code>mapfile</code> command when no array variable name is given
MAILCHECK	How often (in seconds) the shell should check for new mail (The default is 60.)
OLDPWD	The previous working directory used in the shell
OPTERR	If set to 1, the bash shell displays errors generated by the <code>getopts</code> command.
OSTYPE	A string defining the operating system the shell is running on
PIPESTATUS	A variable array containing a list of exit status values from the processes in the foreground process
POSIXLY_CORRECT	If set, bash starts in POSIX mode.
PPID	The process ID (PID) of the bash shell's parent process
PROMPT_COMMAND	If set, the command to execute before displaying the primary prompt
PROMPT_DIRTRIM	An integer used to indicate the number of trailing directory names to display when using the <code>\w</code> and <code>\W</code> prompt string escapes (The directory names removed are replaced with one set of ellipses.)
PS3	The prompt to use for the <code>select</code> command
PS4	The prompt displayed before the command line is echoed if the <code>bash -x</code> parameter is used
PWD	The current working directory

Continues

TABLE 6-2 (continued)

Variable	Description
RANDOM	Returns a random number between 0 and 32767 (Assigning a value to this variable seeds the pseudo-random number generator.)
READLINE_LINE	Readline buffer contents when using <code>bind -x</code> command
READLINE_POINT	Readline buffer content insertion point's current position when using <code>bind -x</code> command
REPLY	The default variable for the <code>read</code> command
SECONDS	The number of seconds since the shell was started (Assigning a value resets the timer to the value.)
SHELL	The full pathname to the bash shell
SHELLOPTS	A colon-separated list of enabled bash shell options
SHLVL	Indicates the shell level, incremented by one each time a new bash shell is started
TIMEFORMAT	A format specifying how the shell displays time values
TMOUT	The value of how long (in seconds) the <code>select</code> and <code>read</code> commands should wait for input (The default of zero indicates to wait indefinitely.)
TMPDIR	Directory name where the bash shell creates temporary files for its use
UID	The numeric real user ID of the current user

You may notice that not all default environment variables are shown when the `set` command is used. When not in use, the default environment variables are not all required to contain a value.

Setting the PATH Environment Variable

When you enter an external command (see Chapter 5) in the shell command line interface (CLI), the shell must search the system to find the program. The `PATH` environment variable defines the directories it searches looking for commands and programs. On this Ubuntu Linux system, the `PATH` environment variable looks like this:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin:/usr/games:/usr/local/games
$
```

This shows that there are eight directories where the shell looks for commands and programs. The directories in the `PATH` are separated by colons.

If a command's or program's location is not included in the `PATH` variable, the shell cannot find it without an absolute directory reference. If the shell cannot find the command or program, it produces an error message:

```
$ myprog
-bash: myprog: command not found
$
```

The problem is that often applications place their executable programs in directories that aren't in the `PATH` environment variable. The trick is ensuring that your `PATH` environment variable includes all the directories where your applications reside.

You can add new search directories to the existing `PATH` environment variable without having to rebuild it from scratch. The individual directories listed in the `PATH` are separated by colons. All you need to do is reference the original `PATH` value and add any new directories to the string. This looks something like this:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:
/sbin:/bin:/usr/games:/usr/local/games
$
$ PATH=$PATH:/home/christine/Scripts
$
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/
games:/usr/local/games:/home/christine/Scripts
$
$ myprog
The factorial of 5 is 120.
$
```

By adding the directory to the `PATH` environment variable, you can now execute your program from anywhere in the virtual directory structure:

```
$ cd /etc
$
$ myprog
The factorial of 5 is 120
$
```

Tip

If you want your program's location to be available to subshells, be sure to export your modified `PATH` environment variable.

A common trick for programmers is to include the single dot symbol in their `PATH` environment variable. The single dot symbol represents the current directory (see Chapter 3):

```
$ PATH=$PATH: .
$
```

```
$ cd /home/christine/Old_Scripts
$
$ myprog2
The factorial of 6 is 720
$
```

Changes to the PATH variable last only until you exit the system or the system reboots. The changes are not persistent. In the next section, you see how you can make changes to environment variables permanent.

Locating System Environment Variables

The Linux system uses environment variables for many purposes. You know now how to modify system environment variables and create your own variables. The trick is in how these environment variables are made persistent.

When you start a bash shell by logging in to the Linux system, by default bash checks several files for commands. These files are called *startup files* or *environment files*. The startup files that bash processes depend on the method you use to start the bash shell. You can start a bash shell in three ways:

- As a default login shell at login time
- As an interactive shell that is started by spawning a subshell
- As a non-interactive shell to run a script

The following sections describe the startup files the bash shell executes in each of these startup methods.

Understanding the login shell process

When you log in to the Linux system, the bash shell starts as a login shell. The login shell typically looks for five different startup files to process commands from:

- /etc/profile
- \$HOME/.bash_profile
- \$HOME/.bashrc
- \$HOME/.bash_login
- \$HOME/.profile

The /etc/profile file is the main default startup file for the bash shell on the system. All users on the system execute this startup file when they log in.

NOTE

Be aware that some Linux distributions use Pluggable Authentication Modules (PAM). In this case, before the bash shell is started, PAM files are processed, including ones that may contain environment variables. PAM file examples include the `/etc/environment` file and the `$HOME/.pam_environment` file. Find more information about PAM at <http://linux-pam.org>.

The other four startup files are specific for each user and can be customized for an individual user's requirements. Let's look closer at these files.

Viewing the `/etc/profile` file

The `/etc/profile` file is the main default startup file for the bash shell. Whenever you log in to the Linux system, bash executes the commands in the `/etc/profile` startup file first. Different Linux distributions place different commands in this file. On this Ubuntu Linux system, the file looks like this:

```
$ cat /etc/profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).

if [ "$PS1" ]; then
  if [ "$BASH" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

# The default umask is now handled by pam_umask.
# See pam_umask(8) and /etc/login.defs.

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
  unset i
fi
$
```

Most of the commands and syntax you see in this file are covered in more detail in Chapter 12 and beyond. Each distribution's `/etc/profile` file has different settings and commands. For example, notice that a file is mentioned in this Ubuntu distribution's `/etc/profile` file above, called `/etc/bash.bashrc`. It contains system environment variables.

However, in this CentOS distribution's `/etc/profile` file listed below, no `/etc/bash.bashrc` file is called. Also note that it sets and exports some system environment variables within itself:

```
$ cat /etc/profile
# /etc/profile

# System wide environment and startup programs, for login setup
# Functions and aliases go in /etc/bashrc

# It's NOT a good idea to change this file unless you know what you
# are doing. It's much better to create a custom.sh shell script in
# /etc/profile.d/ to make custom changes to your environment, to
# prevent the need for merging in future updates.

pathmunge () {
    case ":${PATH}:" in
        *:"$1":*)
            ;;
        *)
            if [ "$2" = "after" ] ; then
                PATH=$PATH:$1
            else
                PATH=$1:$PATH
            fi
    esac
}

if [ -x /usr/bin/id ]; then
    if [ -z "$EUID" ]; then
        # ksh workaround
        EUID=`id -u`
        UID=`id -ru`
    fi
    USER=`id -un`
    LOGNAME=$USER
    MAIL="/var/spool/mail/$USER"
fi

# Path manipulation
if [ "$EUID" = "0" ]; then
    pathmunge /sbin
```

```

        pathmunge /usr/sbin
        pathmunge /usr/local/sbin
    else
        pathmunge /usr/local/sbin after
        pathmunge /usr/sbin after
        pathmunge /sbin after
    fi

    HOSTNAME=`/bin/hostname 2>/dev/null`
    HISTSIZE=1000
    if [ "$HISTCONTROL" = "ignorespace" ] ; then
        export HISTCONTROL=ignoreboth
    else
        export HISTCONTROL=ignoredups
    fi

    export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE HISTCONTROL

    # By default, we want umask to get set. This sets it for login shell
    # Current threshold for system reserved uid/gids is 200
    # You could check uidgid reservation validity in
    # /usr/share/doc/setup-*/uidgid file
    if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
        umask 002
    else
        umask 022
    fi

    for i in /etc/profile.d/*.sh ; do
        if [ -r "$i" ]; then
            if [ "${-#*i}" != "$-" ]; then
                . "$i"
            else
                . "$i" >/dev/null 2>&1
            fi
        fi
    done

    unset i
    unset -f pathmunge
    $

```

Both distributions' `/etc/profile` files use a certain feature. It is a `for` statement that iterates through any files located in the `/etc/profile.d` directory. (for statements are discussed in detail in Chapter 13.) This provides a place for the Linux system to place application-specific startup files that is executed by the shell when you log in. On this Ubuntu Linux system, the following files are in the `profile.d` directory:

```
$ ls -l /etc/profile.d
total 12
-rw-r--r-- 1 root root  40 Apr 15 06:26 appmenu-qt5.sh
-rw-r--r-- 1 root root 663 Apr  7 10:10 bash_completion.sh
-rw-r--r-- 1 root root 1947 Nov 22  2013 vte.sh
$
```

You can see that this CentOS system has quite a few more files in `/etc/profile.d`:

```
$ ls -l /etc/profile.d
total 80
-rw-r--r-- 1 root root 1127 Mar  5 07:17 colorls.csh
-rw-r--r-- 1 root root 1143 Mar  5 07:17 colorls.sh
-rw-r--r-- 1 root root  92 Nov 22  2013 cvs.csh
-rw-r--r-- 1 root root  78 Nov 22  2013 cvs.sh
-rw-r--r-- 1 root root 192 Feb 24 09:24 glib2.csh
-rw-r--r-- 1 root root 192 Feb 24 09:24 glib2.sh
-rw-r--r-- 1 root root  58 Nov 22  2013 gnome-ssh-askpass.csh
-rw-r--r-- 1 root root  70 Nov 22  2013 gnome-ssh-askpass.sh
-rwxr-xr-x 1 root root 373 Sep 23  2009 kde.csh
-rwxr-xr-x 1 root root 288 Sep 23  2009 kde.sh
-rw-r--r-- 1 root root 1741 Feb 20 05:44 lang.csh
-rw-r--r-- 1 root root 2706 Feb 20 05:44 lang.sh
-rw-r--r-- 1 root root 122 Feb  7  2007 less.csh
-rw-r--r-- 1 root root 108 Feb  7  2007 less.sh
-rw-r--r-- 1 root root 976 Sep 23  2011 qt.csh
-rw-r--r-- 1 root root 912 Sep 23  2011 qt.sh
-rw-r--r-- 1 root root 2142 Mar 13 15:37 udisks-bash-completion.sh
-rw-r--r-- 1 root root  97 Apr  5  2012 vim.csh
-rw-r--r-- 1 root root 269 Apr  5  2012 vim.sh
-rw-r--r-- 1 root root 169 May 20  2009 which2.sh
$
```

Notice that several files are related to specific applications on the system. Most applications create two startup files — one for the bash shell (using the `.sh` extension) and one for the `c` shell (using the `.csh` extension).

The `lang.csh` and `lang.sh` files attempt to determine the default language character set used on the system and set the `LANG` environment variable appropriately.

Viewing the `$HOME` startup files

The remaining startup files are all used for the same function — to provide a user-specific startup file for defining user-specific environment variables. Most Linux distributions use only one or two of these four startup files:

- `$HOME/.bash_profile`
- `$HOME/.bashrc`

- \$HOME/.bash_login
- \$HOME/.profile

Notice that all four files start with a dot, making them hidden files (they don't appear in a normal `ls` command listing). Because they are in the user's HOME directory, each user can edit the files and add his or her own environment variables that are active for every bash shell session they start.

NOTE

Environment files are one area where Linux distributions vary greatly. Not every \$HOME file listed in this section exists for every user. For example, some users may have only the \$HOME/.bash_profile file. This is normal.

The first file found in the following ordered list is run, and the rest are ignored:

```
$HOME/.bash_profile
$HOME/.bash_login
$HOME/.profile
```

Notice that \$HOME/.bashrc is not in this list. This is because it is typically run from one of the other files.

TIP

Remember that \$HOME represents a user's home directory. Also, the tilde (~) is used to represent a user's home directory.

This CentOS Linux system contains the following .bash_profile file:

```
$ cat $HOME/.bash_profile
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
$
```

The .bash_profile startup file first checks to see if the startup file, .bashrc, is present in the HOME directory. If it's there, the startup file executes the commands in it.

Understanding the interactive shell process

If you start a bash shell without logging into a system (if you just type `bash` at a CLI prompt, for example), you start what's called an *interactive shell*. The interactive shell doesn't act like the login shell, but it still provides a CLI prompt for you to enter commands.

If bash is started as an interactive shell, it doesn't process the `/etc/profile` file. Instead, it only checks for the `.bashrc` file in the user's `HOME` directory.

On this Linux CentOS distribution, this file looks like this:

```
$ cat .bashrc

# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
$
```

The `.bashrc` file does two things. First, it checks for a common `bashrc` file in the `/etc` directory. Second, it provides a place for the user to enter personal command aliases (discussed in Chapter 5) and private script functions (described in Chapter 17).

Understanding the non-interactive shell process

The last type of shell is a non-interactive subshell. This is the shell where the system can start to execute a shell script. This is different in that there isn't a CLI prompt to worry about. However, you may want to run specific startup commands each time you start a script on your system.

TIP

Scripts can be executed in different ways. Only some execution methods start a subshell. You learn about the different shell execution methods in Chapter 11.

To accommodate that situation, the bash shell provides the `BASH_ENV` environment variable. When the shell starts a non-interactive subshell process, it checks this environment variable for the startup file name to execute. If one is present, the shell executes the file's commands, which typically include variables set for the shell scripts.

On this CentOS Linux distribution, this environment value is not set by default. When a variable is not set, the `printenv` command simply returns the CLI prompt:

```
$ printenv BASH_ENV
$
```

On this Ubuntu distribution, the `BASH_ENV` variable isn't set either. Remember that, when a variable is not set, the `echo` command displays a blank line and returns the CLI prompt:

```
$ echo $BASH_ENV  
  
$
```

So if the `BASH_ENV` variable isn't set, how do the shell scripts get their environment variables? Remember that some shell script execution methods start a subshell, also called a child shell (see Chapter 5). A child shell inherits its parent shell's exported variables.

For example, if the parent shell was a login shell and had variables set and exported in the `/etc/profile` file, `/etc/profile.d/*.sh` files, and the `$HOME/.bashrc` file, the child shell for the script inherits these variables.

However, remember that any variables set but not exported by the parent shell are local variables. Local variables are not inherited by a subshell.

For scripts that do not start a subshell, the variables are already available in the current shell. Thus, even if `BASH_ENV` is not set, both the current shell's local and global variables are present to be used.

Making environment variables persistent

Now that you know your way around the various shell process types and their various environment files, locating the permanent environment variables is much easier. You can also set your own permanent global or local variables using these files.

For global environment variables (those variables needed by all the users on a Linux system), it may be tempting to put new or modified variable settings in the `/etc/profile`, but this is a bad idea. The file could be changed when your distribution is upgraded, and you would lose all the customized variable settings.

It is a better idea to create a file ending with `.sh` in the `/etc/profile.d` directory. In that file, place all your new or modified global environment variable settings.

On most distributions, the best place to store an individual user's persistent bash shell variables is in the `$HOME/.bashrc` file. This is true for all shell process types. However, if the `BASH_ENV` variable is set, keep in mind that unless it points to `$HOME/.bashrc`, you may need to store a user's variables for non-interactive shell types elsewhere.

NOTE

Keep in mind that user environment variables for graphical interface elements, such as the GUI client, may need to be set in different configuration files than where bash shell environment variables are set.

Recall back in Chapter 5 that command `alias` settings are also not persistent. You can also store your personal `alias` settings in the `$HOME/.bashrc` startup file to make them permanent.

Learning about Variable Arrays

A really cool feature of environment variables is that they can be used as *arrays*. An array is a variable that can hold multiple values. Values can be referenced either individually or as a whole for the entire array.

To set multiple values for an environment variable, just list them in parentheses, with values separated by spaces:

```
$ mytest=(one two three four five)
$
```

Not much excitement there. If you try to display the array as a normal environment variable, you'll be disappointed:

```
$ echo $mytest
one
$
```

Only the first value in the array appears. To reference an individual array element, you must use a numerical index value, which represents its place in the array. The numeric value is enclosed in square brackets:

```
$ echo ${mytest[2]}
three
$
```

TIP

Environment variable arrays start with an index value of zero. This can be confusing.

To display an entire array variable, you use the asterisk wildcard character as the index value:

```
$ echo ${mytest[*]}
one two three four five
$
```

You can also change the value of an individual index position:

```
$ mytest[2]=seven
$
$ echo ${mytest[*]}
one two seven four five
$
```

You can even use the `unset` command to remove an individual value within the array, but be careful, because this gets tricky. Watch this example:


```
$ unset mytest[2]
$
$ echo ${mytest[*]}
one two four five
$
$ echo ${mytest[2]}

$ echo ${mytest[3]}
four
$
```

This example uses the `unset` command to remove the value at index value 2. When you display the array, it appears that the other index values just dropped down one. However, if you specifically display the data at index value 2, you see that that location is empty.

Finally, you can remove the entire array just by using the array name in the `unset` command:

```
$ unset mytest
$
$ echo ${mytest[*]}

$
```

Sometimes variable arrays just complicate matters, so they're often not used in shell script programming. They're not very portable to other shell environments, which is a downside if you do lots of shell programming for different shells. Some bash system environment variables use arrays (such as `BASH_VERSINFO`), but overall you probably won't run into them very often.

Summary

This chapter examined the world of Linux environment variables. Global environment variables can be accessed from any child shell spawned by the parent shell in which they're defined. Local environment variables can be accessed only from the process in which they're defined.

The Linux system uses both global and local environment variables to store information about the system environment. You can access this information from the shell command line interface, as well as within shell scripts. The bash shell uses the system environment variables defined in the original Unix Bourne shell, as well as lots of new environment variables. The `PATH` environment variable defines the search pattern the bash shell takes to find an executable command. You can modify the `PATH` environment variable to add your own directories, or even the current directory symbol, to make running your programs easier.

You can also create your own global and local environment variables for your own use. After you create an environment variable, it's accessible for the entire duration of your shell session.

The bash shell executes several startup files when it starts up. These startup files can contain environment variable definitions to set standard environment variables for each bash session. When you log in to the Linux system, the bash shell accesses the `/etc/profile` startup file and three local startup files for each user, `$HOME/.bash_profile`, `$HOME/.bash_login`, and `$HOME/.profile`. Users can customize these files to include environment variables and startup scripts for their own use.

Finally, this chapter discussed the use of environment variable arrays. These environment variables can contain multiple values in a single variable. You can access the values either individually by referencing an index value or as a whole by referencing the entire environment variable array name.

The next chapter dives into the world of Linux file permissions. This is possibly the most difficult topic for novice Linux users. However, to write good shell scripts, you need to understand how file permissions work and be able to use them on your Linux system.