SEVENTH EDITION

"I have found this book to be a very useful classroom text, as well as a great Linux resource. It teaches Linux using a ground-up approach that gives students the chance to progress with their skills and grow into the Linux world. I have often pointed to this book when asked to recommend a solid Linux reference."

—**Eric Hartwell**, Chair, School of Information Technology, ITT Technical Institute

A Practical Guide to Fedora™ and Red Hat® Enterprise Linux®

Includes Full Fedora on DVD

- **The #1 Fedora and RHEL resource** — a tutorial AND on-the-job reference
- **Master Linux administration and security** using the command line, GUI tools, Python, systemd, and **firewalld**
- **Set up key Internet servers,** step by step, including Samba, Apache, MariaDB/MySQL, **sendmail**, OpenSSH, DNS, LDAP, and more
- **Brand-new chapter** on Virtual Machines and Cloud Computing!

Mark G. Sobell

Praise for Previous Editions of *A Practical Guide to Fedora™ and Red Hat® Enterprise Linux®*

"Sobell tackles a massive subject, the vast details of a Linux operating system, and manages to keep the material clear, interesting and engaging. . . . If you want to know how to get the most out of your Red Hat, Fedora, or CentOS system, then this one of the best texts available, in my opinion."

—*Jesse Smith*
Feature Writer for DistroWatch

"Since I'm in an educational environment, I found the content of Sobell's book to be right on target and very helpful for anyone managing Linux in the enterprise. His style of writing is very clear. He builds up to the chapter exercises, which I find to be relevant to real-world scenarios a user or admin would encounter. An IT/IS student would find this book a valuable complement to their education. The vast amount of information is extremely well balanced and Sobell manages to present the content without complicated asides and meandering prose. This is a 'must have' for anyone managing Linux systems in a networked environment or anyone running a Linux server. I would also highly recommend it to an experienced computer user who is moving to the Linux platform."

—*Mary Norbury*
IT Director
Barbara Davis Center
University of Colorado at Denver
from a review posted on slashdot.org

"I had the chance to use your UNIX books when I when was in college years ago at Cal Poly, San Luis Obispo, CA. I have to say that your books are among the best! They're quality books that teach the theoretical aspects and applications of the operating system."

—*Benton Chan*
IS Engineer

"The book has more than lived up to my expectations from the many reviews I read, even though it targets FC2. I have found something very rare with your book: It doesn't read like the standard technical text, it reads more like a story. It's a pleasure to read and hard to put down. Did I say that?! :-)"

—*David Hopkins*
Business Process Architect

```
$ PS2="Input => "
$ who |
Input => grep sam
sam      tty1         2012-05-01 10:37 (:0)
```

PS3: Menu Prompt

The **PS3** variable holds the menu prompt for the **select** control structure (page 1013).

PS4: Debugging Prompt

The **PS4** variable holds the bash debugging symbol (page 995).

IFS: Separates Input Fields (Word Splitting)

The **IFS** (Internal Field Separator) shell variable specifies the characters you can use to separate arguments on a command line. It has the default value of SPACE-TAB-NEWLINE. Regardless of the value of **IFS**, you can always use one or more SPACE or TAB characters to separate arguments on the command line, provided these characters are not quoted or escaped. When you assign character values to **IFS**, these characters can also separate fields—but only if they undergo expansion. This type of interpretation of the command line is called *word splitting* and is discussed on page 411.

Be careful when changing IFS

caution Changing **IFS** has a variety of side effects, so work cautiously. You might find it useful to save the value of **IFS** before changing it. You can then easily restore the original value if a change yields unexpected results. Alternately, you can fork a new shell using a **bash** command before experimenting with **IFS**; if you run into trouble, you can **exit** back to the old shell, where **IFS** is working properly.

The following example demonstrates how setting **IFS** can affect the interpretation of a command line:

```
$ a=w:x:y:z

$ cat $a
cat: w:x:y:z: No such file or directory
$ IFS=":"

$ cat $a
cat: w: No such file or directory
cat: x: No such file or directory
cat: y: No such file or directory
cat: z: No such file or directory
```

The first time cat is called, the shell expands the variable **a**, interpreting the string **w:x:y:z** as a single word to be used as the argument to cat. The cat utility cannot find a file named **w:x:y:z** and reports an error for that filename. After **IFS** is set to a colon (**:**), the shell expands the variable **a** into four words, each of which is an argument to cat. Now cat reports errors for four files: **w**, **x**, **y**, and **z**. Word splitting based on the colon (**:**) takes place only *after* the variable **a** is expanded.

The shell splits all *expanded* words on a command line according to the separating characters found in **IFS**. When there is no expansion, there is no splitting. Consider the following commands:

```
$ IFS="p"
$ export VAR
```

Although **IFS** is set to **p**, the **p** on the **export** command line is not expanded, so the word **export** is not split.

The following example uses variable expansion in an attempt to produce an **export** command:

```
$ IFS="p"
$ aa=export
$ echo $aa
ex ort
```

This time expansion occurs, so the **p** in the token **export** is interpreted as a separator (as the echo command shows). Next, when you try to use the value of the **aa** variable to export the **VAR** variable, the shell parses the **$aa VAR** command line as **ex ort VAR**. The effect is that the command line starts the ex editor with two filenames: **ort** and **VAR**.

```
$ $aa VAR
2 files to edit
"ort" [New File]
Entering Ex mode.  Type "visual" to go to Normal mode.
:q
E173: 1 more file to edit
:q
$
```

If **IFS** is unset, bash uses its default value (SPACE-TAB-NEWLINE). If **IFS** is null, bash does not split words.

Multiple separator characters

tip Although the shell treats sequences of multiple SPACE or TAB characters as a single separator, it treats *each occurrence* of another field-separator character as a separator.

CDPATH: Broadens the Scope of cd

The **CDPATH** variable allows you to use a simple filename as an argument to the cd builtin to change the working directory to a directory other than a child of the working directory. If you typically work in several directories, this variable can speed things up and save you the tedium of using cd with longer pathnames to switch among them.

When **CDPATH** is not set and you specify a simple filename as an argument to cd, cd searches the working directory for a subdirectory with the same name as the argument. If the subdirectory does not exist, cd displays an error message. When **CDPATH** is set, cd searches for an appropriately named subdirectory in the directories in the **CDPATH** list. If it finds one, that directory becomes the working directory. With **CDPATH** set, you can use cd and a simple filename to change the working directory to a child of any of the directories listed in **CDPATH**.

The **CDPATH** variable takes on the value of a colon-separated list of directory pathnames (similar to the **PATH** variable). It is usually set in the **~/.bash_profile** startup file with a command line such as the following:

```
export CDPATH=$HOME:$HOME/literature
```

This command causes cd to search your home directory, the **literature** directory, and then the working directory when you give a cd command. If you do not include the working directory in **CDPATH**, cd searches the working directory if the search of all the other directories in **CDPATH** fails. If you want cd to search the working directory first, include a colon (**:**) as the first entry in **CDPATH**:

```
export CDPATH=:$HOME:$HOME/literature
```

If the argument to the cd builtin is anything other than a simple filename (i.e., if the argument contains a slash [**/**]), the shell does not consult **CDPATH**.

Keyword Variables: A Summary

Table 9-5 lists the bash keyword variables.

Table 9-5 bash keyword variables

Variable	Value
BASH_ENV	The pathname of the startup file for noninteractive shells (page 331)
CDPATH	The cd search path (page 364)
COLUMNS	The width of the display used by **select** (page 1012)
HISTFILE	The pathname of the file that holds the history list (default: **~/.bash_history**; page 377)
HISTFILESIZE	The maximum number of entries saved in **HISTFILE** (default: 1000; page 377)
HISTSIZE	The maximum number of entries saved in the history list (default: 1,000; page 377)
HOME	The pathname of the user's home directory (page 358); used as the default argument for cd and in tilde expansion (page 182)
IFS	Internal Field Separator (page 363); used for word splitting (page 411)
INPUTRC	The pathname of the Readline startup file (default: **~/.inputrc**; page 390)

Table 9-5 bash keyword variables (continued)

Variable	Value
LANG	The locale category when that category is not specifically set using one of the **LC_** variables (page 368)
LC_	A group of variables that specify locale categories including **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, and **LC_NUMERIC**; use the locale builtin (page 369) to display a more complete list including values
LINES	The height of the display used by **select** (page 1012)
MAIL	The pathname of the file that holds a user's mail (page 360)
MAILCHECK	How often, in seconds, bash checks for mail (default: 60; page 360)
MAILPATH	A colon-separated list of file pathnames that bash checks for mail in (page 360)
OLDPWD	The pathname of the previous working directory
PATH	A colon-separated list of directory pathnames that bash looks for commands in (page 359)
PROMPT_COMMAND	A command that bash executes just before it displays the primary prompt
PS1	Prompt String 1; the primary prompt (page 361)
PS2	Prompt String 2; the secondary prompt (page 362)
PS3	The prompt issued by **select** (page 1012)
PS4	The bash debugging symbol (page 995)
PWD	The pathname of the working directory
REPLY	Holds the line that read accepts (page 1042); also used by **select** (page 1012)

Special Characters

Table 9-6 lists most of the characters that are special to the bash shell.

Table 9-6 Shell special characters

Character	Use
NEWLINE	A control operator that initiates execution of a command (page 341)
;	A control operator that separates commands (page 341)
( )	A control operator that groups commands (page 344) for execution by a subshell; these characters are also used to identify a function (page 396)

Table 9-6 Shell special characters (continued)

Character	Use
(( ))	Evaluates an arithmetic expression (pages 408 and 1056)
&	A control operator that executes a command in the background (pages 163 and 342)
|	A control operator that sends standard output of the preceding command to standard input of the following command (pipeline; page 342)
|&	A control operator that sends standard output and standard error of the preceding command to standard input of the following command (page 335)
>	Redirects standard output (page 154)
>>	Appends standard output (page 157)
<	Redirects standard input (page 155)
<<	Here document (page 1014)
*	Matches any string of zero or more characters in an ambiguous file reference (page 166)
?	Matches any single character in an ambiguous file reference (page 165)
\	Quotes the following character (page 142)
'	Quotes a string, preventing all substitution (page 142)
"	Quotes a string, allowing only variable and command substitution (pages 142 and 354)
\`...\`	Performs command substitution [deprecated, see **$()**]
[ ]	Character class in an ambiguous file reference (page 168)
$	References a variable (page 352)
. (dot builtin)	Executes a command in the current shell (page 332)
#	Begins a comment (page 340)
{ }	Surrounds the contents of a function (page 396)
: (null builtin)	Returns *true* (page 1049)
&& (Boolean AND)	A control operator that executes the command on the right only if the command on the left succeeds (returns a zero exit status; page 343)
|| (Boolean OR)	A control operator that executes the command on the right only if the command on the left fails (returns a nonzero exit status; page 343)
! (Boolean NOT)	Reverses the exit status of a command
$()	Performs command substitution (preferred form; page 410)

LPI Locale

In conversational English, a *locale* is a place or location. When working with Linux, a locale specifies the way locale-aware programs display certain kinds of data such as times and dates, money and other numeric values, telephone numbers, and measurements. It can also specify collating sequence and printer paper size.

Localization and internationalization

Localization and internationalization go hand in hand: Internationalization is the process of making software portable to multiple locales while localization is the process of adapting software so that it meets the language, cultural, and other requirements of a specific locale. Linux is well internationalized so you can easily specify a locale for a given system or user. Linux uses variables to specify a locale.

i18n

The term i18n is an abbreviation of the word *internationalization:* the letter *i* followed by 18 letters (*nternationalizatio*) followed by the letter *n*.

l10n

The term l10n is an abbreviation of the word *localization:* the letter *l* followed by 10 letters (*ocalizatio*) followed by the letter *n*.

LPI LC_: Locale Variables

The bash man page lists the following locale variables; other programs use additional locale variables. See the locale man pages (sections 1, 5, and 7) or use the locale **––help** option for more information.

- **LANG**—Specifies the locale category for categories not specified by an **LC_** variable (except see **LC_ALL**). Many setups use only this locale variable and do not specify any of the **LC_** variables.
- **LC_ALL**—Overrides the value of **LANG** and all other **LC_** variables.
- **LC_COLLATE**—Specifies the collating sequence for the sort utility and for sorting the results of pathname expansion (page 354).
- **LC_CTYPE**—Specifies how characters are interpreted and how character classes within pathname expansion and pattern matching behave. Also affects the sort utility when you specify the **–d** (**––dictionary-order**) or **–i** (**––ignore-nonprinting**) options.
- **LC_MESSAGES**—Specifies how affirmative and negative answers appear and the language messages are displayed in.
- **LC_NUMERIC**—Specifies how numbers are formatted (e.g., are thousands separated by a comma or a period?).

You can set one or more of the **LC_** variables to a value using the following syntax:

xx_YY.CHARSET

where ***xx*** is the ISO-639 language code (e.g., **en** = English, **fr** = French, **zu** = Zulu), ***YY*** is the ISO-3166 country code (e.g., **FR** = France, **GF** = French Guiana, **PF** = French Polynesia), and ***CHARSET*** is the name of the character set (e.g., **UTF-8** [page 1279], **ASCII** [page 1237], **ISO-8859-1** [Western Europe]; also called the *character map* or *charmap*). On some systems you can specify ***CHARSET*** using lowercase letters. For