LPI certification 101 exam prep, Part 3

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What does this tutorial cover?

Welcome to "Intermediate administration", the third of four tutorials designed to prepare you for the Linux Professional Institute's 101 exam. In this tutorial, we'll round out your knowledge of fundamental Linux administration skills by covering a variety of topics including: system and Internet documentation, the Linux permissions model, user account management, and login environment tuning.

By the end of this *series* of tutorials (eight in all), you will have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification from the Linux Professional Institute if you so choose.

The LPI logo is a trademark of Linux Professional Institute.

Should I take this tutorial?

This tutorial (Part 3) is ideal for those who want to learn about the Linux permissions model and account management, as well as system and Internet documentation. For some, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way of "rounding out" their foundational Linux system administration skills.

If you are new to Linux, we recommend that you complete *Part 1* and *Part 2* of this tutorial series before continuing.

About the authors

For technical questions about the content of this tutorial, contact the authors:
  * Daniel Robbins, at *drobbins@gentoo.org*
  * Chris Houser, at *chouser@gentoo.org*
  * Aron Griffis, at *agriffis@gentoo.org*

Daniel Robbins lives in Albuquerque, New Mexico, and is the President/CEO of *Gentoo Technologies, Inc.*, the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at SONY Electronic Publishing/Psygnosis. Daniel enjoys spending time with his wife, Mary, and their daughter, Hadassah.

Chris Houser, known to many of his friends as "Chouser", has been a UNIX proponent since 1994 when joined the administration team for the computer science network at Taylor University in Indiana, where he earned his Bachelor's degree in Computer Science and Mathematics. Since then, he has gone on to work in Web application programming, user interface design, professional video software support, and now Tru64 UNIX device driver

programming at *Compaq*. He has also contributed to various free software projects, most recently to *Gentoo Linux*. He lives with his wife and two cats in New Hampshire.

Aron Griffis graduated from Taylor University with a degree in Computer Science and an award that proclaimed him the "Future Founder of a Utopian UNIX Commune". Working towards that goal, Aron is employed by *Compaq* writing network drivers for Tru64 UNIX, and spending his spare time plunking out tunes on the piano or developing *Gentoo Linux*. He lives with his wife Amy (also a UNIX engineer) in Nashua, NH.

Section 2. System and network documentation

Types of Linux system documentation

There are essentially three sources of documentation on a Linux system: manual pages, info pages, and application-bundled documentation in /usr/share/doc. In this section, we'll find out how to explore each of these sources before looking "outside the box" for more information.

Manual pages

Manual pages, or "man pages", are the classic form of UNIX and Linux reference documentation. Ideally, you can look up the man page for any command, configuration file, or library routine. In practice, Linux is free software and some pages haven't been written or are showing their age. Nonetheless, man pages are the first place to look when you need help.

To access a man page, simply type **man** followed by your topic of inquiry. A pager will be started, so you will need to press **q** when you're done reading. For example, to look up information about the **ls** command, you would type:

```
$ man ls
```

Manual pages, continued

Knowing the layout of a man page can be helpful to jump quickly to the information you need. In general, you will find the following sections in a man page:

NAME	Name and one-line description of the command
SYNOPSIS	How to use the command
DESCRIPTION	In-depth discussion on the functionality of the command
EXAMPLES	Suggestions for how to use the command
SEE ALSO	Related topics (usually man pages)

---

man page sections

The files that comprise manual pages are stored in /usr/share/man (or in /usr/man on some older systems). Inside that directory, you will find the manual pages are organized into the following sections:

man1	User programs
man2	System calls
man3	Library functions
man4	Special files
man5	File formats
man6	Games
man7	Miscellaneous

Multiple man pages

Some topics exist in more than one section. To demonstrate this, let's use the **whatis** command, which shows all the available man pages for a topic:

```
$ whatis printf
printf              (1)  - format and print data
printf              (3)  - formatted output conversion
```

In this case, **man printf** would default to the page in section 1 ("User Programs"). If we were writing a C program, we might be more interested in the page from section 3 ("Library functions"). You can call up a man page from a certain section by specifying it on the command line, so to ask for *printf(3)*, we would type:

```
$ man 3 printf
```

Finding the right man page

Sometimes it's hard to find the right man page for a given topic. In that case, you might try using **man -k** to search the **NAME** section of the man pages. Be warned that it's a substring search, so running something like **man -k ls** will give you a lot of output! Here's an example using a more specific query:

```
$ man -k whatis
apropos             (1)  - search the whatis database for strings
makewhatis          (8)  - Create the whatis database
whatis              (1)  - search the whatis database for complete words
```

All about apropos

Ah, the example on the previous panel brings up a couple more points! First, the **apropos** command is exactly equivalent to **man -k**. (In fact, I'll let you in on a little secret. When you run **man -k**, it actually runs **apropos** behind the scenes.) The second point is the **makewhatis** command, which scans all the man pages on your Linux system and builds the database for **whatis** and **apropos**. Usually this is run periodically by root to keep the database updated:

```
# makewhatis
```

For more information on "man" and friends, you should start with its man page:

```
$ man man
```

The MANPATH

By default, the **man** program will look for man pages in /usr/share/man, /usr/local/man, /usr/X11R6/man, and possibly /opt/man. Sometimes, you may find that you need to add an

additional item to this search path. If so, simply edit /etc/man.conf in a text editor and add a line that looks like this:

```
MANPATH /opt/man
```

From that point forward, any man pages in the **/opt/man/man*** directories will be found. Remember that you'll need to rerun **makewhatis** to add these new man pages to the whatis database.

GNU info

One shortcoming of man pages is that they don't support hypertext, so you can't jump easily from one to another. The GNU folks recognized this shortcoming, so they invented another documentation format: "info" pages. Many of the GNU programs come with extensive documentation in the form of info pages. You can start reading info pages with the "info" command:

```
$ info
```

Calling **info** in this way will bring up an index of the available pages on the system. You can move around with the arrow keys, follow links (indicated with a star) using the enter key, and quit by pressing **q**. The keys are based on Emacs, so you should be able to navigate easily if you're familiar with that editor.

GNU info, continued

You can also specify an info page on the command line:

```
$ info diff
```

For more information on using the **info** reader, try reading its info page. You should be able to navigate primitively using the few keys I've already mentioned:

```
$ info info
```

/usr/share/doc

There is a final source for help within your Linux system. Many programs are shipped with additional documentation in other formats: text, PDF, PostScript, HTML, to name a few. Take a look in /usr/share/doc (or /usr/doc on older systems). You'll find a long list of directories, each of which came with a certain application on your system. Searching through this documentation can often reveal some gems that aren't available as man pages or info pages, such as tutorials or additional technical documentation. A quick check reveals there's a lot of reading material available:

```
$ cd /usr/share/doc
$ find . -type f | wc -l
```

```
7582
```

Whew! Your homework this evening is to read just half (3791) of those documents. Expect a quiz tomorrow. ;-)

The Linux Documentation Project

In addition to system documentation, there are number of excellent Linux resources on the Internet. The Linux Documentation Project is a group of volunteers who are working on putting together the complete set of free Linux documentation. This project exists to consolidate various pieces of Linux documentation into a location that is easy to search and use. You can check out the LDP at: *http://www.linuxdoc.org/*

An LDP overview

The LDP is made up of the following areas:

* Guides - longer, more in-depth books, such as *The Linux Programmer's Guide*
* HOWTOs - subject-specific help, such as the *DSL HOWTO*
* FAQs - Frequently Asked Questions with answers, such as the *Brief Linux FAQ*
* man pages - help on individual commands (these are the same manual pages you get on your Linux system when you use the **man** command)

If you aren't sure which section to peruse, you can take advantage of the search box, which allows you to find things by topic.

The LDP additionally provides a list of Links and Resources such as *Linux Gazette* and *LinuxFocus*, as well links to mailing lists and news archives.

Mailing lists

Mailing lists provide probably the most important point of collaboration for Linux developers. Often projects are developed by contributors who live far apart, possibly even on opposite sides of the globe. Mailing lists provide a method for each developer on a project to contact all the others, and to hold group discussions via e-mail. One of the most famous development mailing lists is the "Linux Kernel Mailing List", which is described at *http://www.tux.org/lkml/*.

More about mailing lists

In addition to development, mailing lists can provide a method for asking questions and receiving answers from knowledgeable developers, or even other users. For example, individual distributions often provide mailing lists for newcomers. You can check your distribution's Web site for information on the mailing lists it provides.

If you took the time to read the LKML FAQ at the link on the previous panel, you might have

noticed that mailing list subscribers often don't take kindly to questions being asked repeatedly. It's always wise to search the archives for a given mailing list before writing your question. Chances are, it will save you time, too!

Newsgroups

Internet "newsgroups" are similar to mailing lists, but are based on a protocol called NNTP ("Network News Transfer Protocol") instead of using e-mail. To participate, you need to use an NNTP client such as **slrn** or **pan**. The primary advantage is that you only take part in the discussion when you want, instead of constantly having it arrive in your inbox. :-)

The newsgroups of primary interest start with **comp.os.linux**. You can browse the list on the LDP site at *http://www.linuxdoc.org/linux/#ng*.

As with mailing lists, newsgroup discussion is often archived. A popular newsgroup archiving site is *Deja News*.

Vendor and third-party Web sites

Web sites for the various Linux distributions often provide updated documentation, installation instructions, hardware compatibility/incompatibility statements, and other support such as a knowledge base search tool. For example:

* *Redhat Linux*
* *Debian Linux*
* *Gentoo Linux*
* *SuSE Linux*
* *Caldera*
* *Turbolinux*

Linux consultancies

Some Linux consultancies, such as Linuxcare and Mission Critical Linux, provide some free documentation as well as pay-for support contracts. There are many Linux consultancies; below are a couple of the larger examples:

* *LinuxCare*
* *Mission Critical Linux*

Hardware and software vendors

Many hardware and software vendors have added Linux support to their products in recent years. At their sites, you can find information about which hardware supports Linux, software development tools, released sources, downloads of Linux drivers for specific hardware, and other special Linux projects. For example:

* *IBM and Linux*
* *Compaq and Linux*
* *SGI and Linux*
* *HP and Linux*
* *Sun and Linux*
* *Oracle and Linux*

Developer resources

In addition, many hardware and software vendors have developed wonderful resources for Linux developers and administrators. At the risk of sounding self-promoting, one of the most valuable Linux resources run by a hardware/software vendor is the *IBM developerWorks Linux zone*.

Section 3. The Linux permissions model

One user, one group

In this section, we'll take a look at the Linux permissions and ownership model. We've already seen that every file is owned by one user and one group. This is the very core of the permissions model in Linux. You can view the user and group of a file in a **ls -l** listing:

```
$ ls -l /bin/bash
-rwxr-xr-x    1 root      wheel       430540 Dec 23 18:27 /bin/bash
```

In this particular example, the **/bin/bash** executable is owned by **root** and is in the **wheel** group. The Linux permissions model works by allowing three independent levels of permission to be set for each filesystem object -- those for the file's owner, the file's group, and for all other users.

Understanding "ls -l"

Let's take a look at our **ls -l** output and inspect the first column of the listing:

```
$ ls -l /bin/bash
-rwxr-xr-x    1 root      wheel       430540 Dec 23 18:27 /bin/bash
```

This first field **-rwxr-xr-x** contains a *symbolic* representation of this particular files' permissions. The first character (**-**) in this field specifies the *type* of this file, which in this case is a regular file. Other possible first characters:

```
'd'  directory
'l'  symbolic link
'c'  character special device
'b'  block special device
'p'  fifo
's'  socket
```

Three triplets

```
$ ls -l /bin/bash
-rwxr-xr-x    1 root      wheel       430540 Dec 23 18:27 /bin/bash
```

The rest of the field consists of *three* character triplets. The first triplet represents permissions for the owner of the file, the second represents permissions for the file's group, and the third represents permissions for all other users:

```
"rwx"
"r-x"
"r-x"
```

Above, the **r** means that reading (looking at the data in the file) is allowed, the **w** means that writing (modifying the file, as well as deletion) is allowed, and the **x** means that 'execute'

(running the program) is allowed. Putting together all this information, we can see that everyone is able to read the contents of and execute this file, but only the owner (root) is allowed to modify this file in any way. So, while normal users can copy this file, only root is allowed to update it or delete it.

Who am I?

Before we take a look at how to change the user and group ownership of a file, let's first take a look at how to learn your current user id and group membership. Unless you've used the **su** command recently, your current user id is the one you used to login to the system. If you use **su** frequently, however, you may not remember your current effective user id. To view it, type **whoami**:

```
# whoami
root
# su drobbins
$ whoami
drobbins
```

What groups am I in?

To see what groups you belong to, use the **group** command:

```
$ groups
drobbins wheel audio
```

In the above example, I'm a member of the **drobbins**, **wheel** and audio groups. If you want to see what groups other user(s) are in, specify their usernames as arguments:

```
$ groups root daemon
root : root bin daemon sys adm disk wheel floppy dialout tape video
daemon : daemon bin adm
```

Changing user and group ownership

To change the owner or group of a file or other filesystem object, use **chown** or **chgrp** respectively. Each of these commands takes a name followed by one or more filenames.

```
# chown root /etc/passwd
# chgrp wheel /etc/passwd
```

You can also set the owner and group simultaneously with an alternate form of the chown command:

```
# chown root.wheel /etc/passwd
```

You may not use **chown** unless you are the superuser, but **chgrp** can be used by anyone to change the group ownership of a file to a group to which they belong.

Recursive ownership changes

Both chown and **chgrp** have a **-R** option that can be used to tell them to recursively apply ownership and group changes to an entire directory tree. For example:

```
# chown -R drobbins /home/drobbins
```

Introducing chmod

chown and **chgrp** can be used to change the owner and group of a filesystem object, but another program -- called **chmod** -- is used to change the **rwx** permissions that we can see in an **ls -l** listing. chmod takes two or more arguments: a "mode", describing how the permissions should be changed, followed by a file or list of files that should be affected:

```
$ chmod +x scriptfile.sh
```

In the above example, our "mode" is **+x**. As you might guess, a **+x** mode tells **chmod** to make this particular file executable for both the user, group and for anyone else.

If we wanted to *remove* all execute permissions of a file, we'd do this:

```
$ chmod -x scriptfile.sh
```

User/group/other granularity

So far, our **chmod** examples have affected permissions for all three triplets -- the user, the group, and all others. Often, it's handy to modify only one or two triplets at a time. To do this, simply specify the symbolic character for the particular triplets you'd like to modify before the **+** or **-** sign. Use **u** for the "user" triplet, **g** for the "group" triplet, and **o** for the "other/everyone" triplet:

```
$ chmod go-w scriptfile.sh
```

We just removed write permissions for the group and all other users, but left "owner" permissions untouched.

Resetting permissions

In addition to flipping permission bits on and off, we can also reset them altogether. By using the **=** operator, we can tell **chmod** that we want the specified permissions and no others:

```
$ chmod =rx scriptfile.sh
```

Above, we just set all "read" and "execute" bits, and unset all "write" bits. If you just want to reset a particular triplet, you can specify the symbolic name for the triplet before the **=** as

follows:

```
$ chmod u=rx scriptfile.sh
```

Numeric modes

Up until now, we've used what are called "symbolic" modes to specify permission changes to **chmod**. However, there's another commonly-used way of specifying permissions -- using a 4-digit octal number. Using this syntax, called numeric permissions syntax, each digit represents a permissions triplet. For example, in **1777**, the **777** set the 'owner', 'group', and 'other' flags that we've been discussing through this section. The **1** is used to set the special permissions bits, which we'll cover at the end of this section. This chart shows how the second through fourth digits (**777**) are interpreted:

mode	digit
rwx	7
rw-	6
r-x	5
r--	4
-wx	3
-w-	2
--x	1
---	0

Numeric permission syntax

Numeric permission syntax is especially useful when you need to specify *all* permissions for a file, such as in the following example:

```
$ chmod 0755 scriptfile.sh
$ ls -l scriptfile.sh
-rwxr-xr-x    1 drobbins drobbins          0 Jan  9 17:44 scriptfile.sh
```

In this example, we used a mode of **0755**, which expands to a complete permissions setting of "-rwxr-xr-x".

The umask

When a process creates a new file, it specifies the permissions that it would like the new file to have. Often, the mode requested is **0666** (readable and writable by everyone), which is more permissive that we would like. Fortunately, Linux consults something called a "umask" whenever a new file is created. The system uses the umask value to reduce the originally-specified permissions to something more reasonable and secure. You can view your current umask setting by typing **umask** at the command line:

```
$ umask
0022
```

On Linux systems, the umask normally defaults to **0022**, which allows others to read your new files (if they can get to them) but not modify them.

The umask, continued

To make new files more secure by default, you can change the umask setting:

```
$ umask 0077
```

This umask will make sure that the group and others will have absolutely no permissions for any newly-created files. So, how does the umask work? Unlike "regular" permissions on files, the umask specifies which permissions should be turned *off*. Let's consult our mode-to-digit mapping table so that we can understand what a umask of **0077** means:

mode	digit
rwx	7
rw-	6
r-x	5
r--	4
-wx	3
-w-	2
--x	1
---	0

Using our table, the last three digits of **0077** expand to **---rwxrwx**. Now, remember that the umask tells the system which permissions to *disable*. Putting two and two together, we can see that all "group" and "other" permissions will be turned off, while "user" permissions will remain untouched.

Introducing suid and sgid

When you initially log in, a new shell process is started. You already know that, but what you may not know is that this new shell process (typically **bash**) runs using your user id. As such, the **bash** program can access all files and directories that you own. In fact, we as users we are totally dependent on other programs to perform operations on our behalf. Because the programs you start inherit *your* user id, they cannot access any filesystem objects for which you haven't been granted access.

Introducing suid and sgid, continued

For example, the passwd file cannot be changed by normal users directly, because 'write' flag is off for every user except 'root':

```
$ ls -l /etc/passwd
-rw-r--r--    1 root      wheel         1355 Nov  1 21:16 /etc/passwd
```

However, normal users *do* need to be able to modify **/etc/passwd** (at least indirectly) whenever they need to change their password. But, if the user is unable to modify this file,

how exactly does this work?

suid

Thankfully, the Linux permissions model has two special bits called "suid" and "sgid". When an executable program has the "suid" bit set, it will run on behalf of the *owner* of the executable, rather than on behalf of the person who started the program.

Now, back to the **/etc/passwd** problem. If we take a look at the **passwd** executable, we can see that it's owned by root:

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x    1 root      wheel        17588 Sep 24 00:53 /usr/bin/passwd
```

You'll also note that in place of an **x** in the user's permission triplet, there's an **s**. This indicates that, for this particular program, the suid and executable bits are set. Because of this, when **passwd** runs, it will execute on behalf of the **root** user (with full superuser access) rather than that of the user who ran it. And because **passwd** runs with **root** access, it's able to modify the **/etc/passwd** file with no problem.

suid/sgid caveats

We've seen how suid works, and sgid works in a similar way. It allows programs to inherit the group ownership of the program rather than that of the current user.

Here's some miscellaneous yet important information about suid and sgid. First, suid and sgid bits occupy the same space as the **x** bits in a **ls -l** listing. If the **x** bit is also set, the respective bits will show up as **s** (lowercase). However, if the **x** bit is not set, it will show up as a **S** (uppercase).

Another important note: suid and suid come in handy in many circumstances, but improper use of these bits can allow the security of a system to be breached. It's best to have as few 'suid' programs as possible. The passwd command is one of the few that must be 'suid'.

Changing suid and sgid

Setting and removing the suid and sgid bits is fairly straightforward. Here, we set the suid bit:

```
# chmod u+s /usr/bin/myapp
```

And here, we *remove* the sgid bit from a directory. We'll see how the sgid bit affects directories in just a few panels:

```
# chmod g-s /home/drobbins
```

Permissions and directories

So far, we've been looking at permissions from the perspective of regular files. When it comes to directories, things are a bit different. Directories use the same permissions flags, but they are interpreted to mean slightly different things.

For a directory, if the 'read' flag is set, you may *list* the contents of the directory; 'write' means you may *create* files in the directory, and 'execute' means you may *enter* the directory and access any sub-directories inside. Without the 'execute' flag, the filesystem objects inside a directory aren't accessible. Without a 'read' flag, the filesystem objects inside a directory aren't viewable, but objects inside the directory can still be accessed as long as someone knows the full path to the object on disk.

Directories and sgid

And, if a directory has the 'sgid' flag enabled, any filesystem objects created inside it will inherit the group of the directory. This particular feature comes in handy when you need to create a directory tree to be used by a group of people that all belong to the same group. Simply do this:

```
# mkdir /home/groupspace
# chgrp mygroup /home/groupspace
# chmod g+s /home/groupspace
```

Now, any users in the group **mygroup** can create files or directories inside **/home/groupspace** and they will be automatically assigned a group ownership of **mygroup** as well. Depending on the users' umask setting, new filesystem objects may or may not be readable, writable or executable by other members of the **mygroup** group.

Directories and deletion

By default, Linux directories behave in a way that may not be ideal in all situations. Normally, anyone can rename or delete a file inside a directory, as long as they have *write* access to that directory. For directories used by individual users, this behavior is usually just fine.

However, for directories that are used by many users, especially **/tmp** and **/var/tmp**, this behavior can be bad news. Since *anyone* can write to these directories, *anyone* can delete or rename anyone else's files -- even if they don't own them! Obviously, it's hard to use /tmp for anything meaningful when any other user can type "rm -rf /tmp/*" at any time and destroy everyone's files.

Thankfully, Linux has something called the "sticky bit". When **/tmp** has the sticky bit set (with a **chmod +t**), the only people who are able to delete or rename files in **/tmp** are the directory's owner (typically **root**) the file's owner, or **root**. Virtually all Linux distributions enable **/tmp**'s sticky bit by default, but you may find that the sticky bit comes in handy in other situations.

The elusive first digit

And to conclude this section, we finally take a look at the elusive first digit of a numeric

mode. As you can see, this first digit is used for setting the sticky, suid and sgid bits:

suid	sgid	sticky	mode digit
on	on	on	7
on	on	off	6
on	off	on	5
on	off	off	4
off	on	on	3
off	on	off	2
off	off	on	1
off	off	off	0

Here's an example of how to use a 4-digit numeric mode to set permissions for a directory that will be used by a workgroup:

```
# chmod 1775 /home/groupfiles
```

As homework, figure out what the meaning of the **1755** numeric permissions setting. :)

Section 4. Linux account management

Introducing /etc/passwd

In this section, we'll take a look at the Linux account management mechanism. I'll start by introducing the **/etc/passwd** file, which defines all the users that exist on a Linux system. You can view your own **/etc/passwd** file by typing "less **/etc/passwd**.

Each line in **/etc/passwd** defines a user account. Here's an example line from my **/etc/passwd** file:

```
drobbins:x:1000:1000:Daniel Robbins:/home/drobbins:/bin/bash
```

As you can see, there is quite a bit of information on this line. In fact, each **/etc/passwd** line consists of multiple fields, each separated by a **:**.

The first field defines the username (**drobbins)**), and the second field contains an **x**. On ancient Linux systems, this field would contain an encrypted password to be used for authentication, but virtually all Linux systems now store this password information in another file.

The third field (**1000**) defines the numeric user id associated with this particular user, and the fourth field (**1000**) associates this user with a particular group; in a few panels, we'll see where group **1000** is defined.

The fifth field contains a textual description of this account -- in this case, the user's name. The sixth field defines this user's home directory, and the seventh field specifies the user's default shell -- the one that will be automatically started when this user logs in.

/etc/passwd tips and tricks

You've probably noticed that there are many more user accounts defined in **/etc/passwd** than actually log in to your system. This is because various Linux components use user accounts to enhance security. Typically, these system accounts have a user id ("uid") of under 100, and many of them will have something like **/bin/false** listed as a default shell. Since the **/bin/false** program does nothing but exit with an error code, this effectively prevents these accounts from being used as login accounts -- they are for internal use only.

/etc/shadow

So, user accounts themselves are defined in **/etc/passwd**. Linux systems contain a companion file to **/etc/passwd** that's called **/etc/shadow**. This file, unlike **/etc/passwd**, is readable only by **root** and contains encrypted password information. Let's look at a sample line from **/etc/shadow**:

```
drobbins:$1$12345678901234567890123456789901:11664:0:-1:-1:-1:-1:0
```

Each line defines password information for a particular account, and again, each field is

separated by a **:**. The first field defines the particular user account with which this shadow entry is associated. The second field contains an encrypted password. The remaining fields are described in the following table:

field 3	Number of days since 1/1/1970 that the password was modified
field 4	# of days before the password will be allowed to be changed (0 for "change anytime")
field 5	# of days before system will force user to change to a new password (-1 for "never")
field 6	# of days before password expires that user will be warned about expiration (-1 for "no warning")
field 7	# of days after password expiration that this account is automatically disabled by the system (-1 for "never disable")
field 8	# of days that this account has been disabled (-1 for "this account is enabled")
field 9	Reserved for future use

/etc/group

Next, we take a look at the **/etc/group** file, which defines all the groups on a Linux system. Here's a sample line:

```
drobbins:x:1000:
```

The **/etc/group** field format is as follows. The first field defines the name of the group; the second field is a vestigial password field that now simply holds an **x**, and the third field defines the numeric group id of this particular group. The fourth field (empty in the above example) defines any users that are members of this group.

You'll recall that our sample **/etc/passwd** line referenced a group id of 1000. This has the effect of placing the **drobbins** user in the **drobbins** group, even though the **drobbins** username isn't listed in the fourth field of **/etc/group**.

Group notes

A note about associating users with groups -- on some systems, you'll find that every new login account is associated with an identically-named (and usually identically-numbered) group. On other systems, all login accounts will belong to a single **users** group. The approach that you use on the system(s) you administrate is up to you. Creating matching groups for each user has the advantage of allowing users to more easily control access to their own files by placing trusted friends in their personal group.

Adding a user and group by hand

Now, I'll show you how to create your own user and group account. The best way to learn how to do this is to add a new user to the system *manually*. To begin, first make sure that your **EDITOR** environment variable is set to your favorite text editor:

```
# echo $EDITOR
vim
```

If it isn't, you can set EDITOR by typing something like:

```
# export EDITOR=/usr/bin/emacs
```

Now, type:

```
# vipw
```

You should now find yourself in your favorite text editor with the **/etc/passwd** file loaded up on the screen. When modifying system passwd and group files, it's very important to use the **vipw** and **vigr** commands. They take extra precautions to ensure that your critical **passwd** and **group** files are locked properly so they don't become corrupted.

Editing /etc/passwd

Now that you have the **/etc/passwd** file up, go ahead and add the following line:

```
testuser:x:3000:3000:LPI tutorial test user:/home/testuser:/bin/false
```

We've just added a "testuser" user with a UID of 3000. We've added him to a group with a GID of 3000, which we haven't created just yet. Alternatively, we could have assigned this user to the GID of the **users** group if we wanted. This new user has a comment that reads **LPI tutorial test user**; the user's home directory is set to **/home/testuser**, and the user's shell is set to **/bin/false** for security purposes. If we were creating an non-test account, we would set the shell to **/bin/bash**. OK, go ahead and save your changes and exit.

Editing /etc/shadow

Now, we need to add an entry in /etc/shadow for this particular user. To do this, type **vipw -s**. You'll be greeted with your favorite editor, which now contains the **/etc/shadow** file. Now, go ahead and *copy* the line of an existing user account (i.e. has a password and is longer than the standard system account entries):

```
drobbins:$1$12345678901234567890123456789012345678901:11664:0:-1:-1:-1:-1:0
```

Now, change the username on the copied line to the name of your new user, and ensure that all fields (particularly the password aging ones) are set to your liking:

```
testuser:$1$12345678901234567890123456789012345678901:11664:0:-1:-1:-1:-1:0
```

Now, save and exit.

Setting a password

You'll be back at the prompt. Now, it's time to set a password for your new user:

```
# passwd testuser
Enter new UNIX password: (enter a password for testuser)
Retype new UNIX password: (enter testuser's new password again)
```

Editing /etc/group

Now that **/etc/passwd** and **/etc/shadow** are set up, it's now time to get **/etc/group** configured properly. To do this, type:

```
# vigr
```

Your **/etc/group** file will appear in front of you, ready for editing. Now, if you chose to assign a default group of **users** for your particular test user, you do not need to add any groups to **/etc/groups**. However, if you chose to create a new group for this user, go ahead and add the following line:

```
testuser:x:3000:
```

Now save and exit.

Creating a home directory

We're nearly done. Type the following commands to create **testuser**'s home directory:

```
# cd /home
# mkdir testuser
# chown testuser.testuser testuser
# chmod o-rwx testuser
```

Our user's home directory is now in place and the account is ready for use. Well, almost ready. If you'd like to use this account, you'll need to use **vipw** to change testuser's default shell to **/bin/bash** so that the user can log in.

Account admin utils

Now that you know how to add a new account and group by hand, I'm going to review the various time-saving account administration utilities available under Linux. Due to space constraints, I won't be going into a lot of detail describing these commands. Remember that you can always get more information about a command by viewing the command's man page. If you are planning to take the LPIC 101 exam, I recommend that you spend some time familiarizing yourself with each of these commands.

newgrp

By default, any files that a user creates are assigned to the user's group specified in **/etc/passwd**. If the user belongs to other groups, he or she can type **newgrp thisgroup** to set current default group membership to the group **thisgroup**. Then, any new files created

will inherit this group membership.

chage

The **chage** command is used to view and change the password aging setting stored in **/etc/shadow**

gpasswd

A general-purpose group administration tool

groupadd/groupdel/groupmod

Used to add/delete/modify groups in /etc/group

More commands

useradd/userdel/usermod

Used to add/delete/modify users in **/etc/passwd**. These commands also perform various other convenience functions. See man pages for more information.

pwconv/grpconv

Used to convert **passwd** and **group** files to "new-style" shadow passwords. Virtually all Linux systems already use shadow passwords, so you should never need to use these commands.

pwunconv/grpunconv

Used to convert **passwd**, **shadow** and **group** files to "old-style" non-shadow passwords. You should never need to use these commands.

Section 5. Tuning the user environment

Introducing "fortune"

Your shell has many useful options that can be set to fit your personal preferences. So far, however, we haven't discussed any way to have these settings set up automatically every time you log in, except for re-typing them each time. In this section we will look at tuning your login environment by modifying startup files.

First, let's add a friendly message for when you first log in. To see an example message, run **fortune**:

```
$ fortune
No amount of careful planning will ever replace dumb luck.
```

.bash_profile

Now, let's set up **fortune** so that it gets run every time you log in. Use your favorite text editor to edit a file named **.bash_profile** in your home directory. If the file doesn't exist already, go ahead and create it. Insert a line at the top:

```
fortune
```

Try logging out and back in. Unless you're running a display manager like **xdm**, **gdm** or **kdm**, you should be greeted cheerfully when you log in:

```
mycroft.flatmonk.org login: chouser
Password:
Freedom from incrustations of grime is contiguous to rectitude.
$
```

The login shell

When bash started, it walked through the **.bash_profile** file in your home directory, running each line as though it had been typed at a bash prompt. This is called "sourcing" the file.

Bash acts somewhat differently depending on how it is started. If it is started as a "login" shell, it will act as it did above -- first sourcing the system-wide **/etc/profile**, and then your personal **~/.bash_profile**.

There are two ways to tell bash to run as a login shell. One way is used when you first log in -- bash is started with a process name of **-bash**. You can see this in your process listing:

```
$ ps u
USER       PID %CPU %MEM   VSZ   RSS TTY       STAT START   TIME COMMAND
chouser    404  0.0  0.0  2508   156 tty2       S     2001   0:00 -bash
```

You will probably see a much longer listing, but you should have at least one **COMMAND**

with a dash before the name of your shell, like **-bash** in the example above. This dash is used by the shell to determine if it's being run as a 'login' shell.

Understanding --login

The second way to tell bash to run as a "login" shell is with the **--login** command-line option. This is sometimes used by terminal emulators (like **xterm**) to make their bash sessions act like initial login sessions.

After you have logged in, more copies of your shell will be run. Unless they are started with **--login** or have a dash in the process name, these sessions will not be "login" shells. If they give you a prompt, however, they are called "interactive" shells. If bash is started as "interactive", but not "login", it will ignore **/etc/profile** and **~/.bash_profile** and will instead source **~/.bashrc**.

interactive	login	profile	rc
yes	yes	source	ignore
yes	no	ignore	source
no	yes	source	ignore
no	no	ignore	ignore

Testing for interactivity

Sometimes bash sources your **~/.bashrc**, even though it isn't really interactive, such as when using commands like **rsh** and **scp**. This is important to keep in mind because printing out text, like we did with the **fortune** command earlier, can really mess up these non-interactive bash sessions. It's a good idea to use the **PS1** variable to detect if the current shell is truly interactive before printing text from a startup file:

```
if [ -n "$PS1" ]; then
        fortune
fi
```

/etc/profile and /etc/skel

As a system administrator, you are in charge of **/etc/profile**. Since it is sourced by everyone when they first log in, it is important to keep it in working order. It is also a powerful tool in making things work correctly for new users as soon as they log into their new account.

However, there are some settings that you may want new users to have as defaults, but also allow them to change easily. This is where the **/etc/skel** directory comes in. When you use the **useradd** command to create a new user account, it copies all the files from **/etc/skel** into the user's new home directory. That means you can put helpful **.bash_profile** and **.bashrc** files in **/etc/skel** to get new users off to a good start.

export

Variables in bash can be marked so that they are set the same in any new shells that it starts; this is called being marked for **export**. You can have bash list all of the variables that are currently marked for export in your shell session:

```
$ export
declare -x EDITOR="vim"
declare -x HOME="/home/chouser"
declare -x MAIL="/var/spool/mail/chouser"
declare -x PAGER="/usr/bin/less"
declare -x PATH="/bin:/usr/bin:/usr/local/bin:/home/chouser/bin"
declare -x PWD="/home/chouser"
declare -x TERM="xterm"
declare -x USER="chouser"
```

Marking variables for export

If a variable is not marked for export, any new shells that it starts will not have that variable set. However, you can mark a variable for export by passing it to the **export** built-in:

```
$ FOO=foo
$ BAR=bar
$ export BAR
$ echo $FOO $BAR
foo bar
$ bash
$ echo $FOO $BAR
bar
```

In this example, the variables **FOO** and **BAR** were both set, but only **BAR** was marked for export. When a new bash was started, it had lost the value for **FOO**. If you exit this new bash, you can see that the original one still has values for both **FOO** and **BAR**:

```
$ exit
$ echo $FOO $BAR
foo bar
```

Export and set -x

Because of this behavior, variables can be set in **~/.bash_profile** or **/etc/profile** and marked for export, and then never need to be set again. There are some options that cannot be exported, however, and so they must be put in put in your **~/.bashrc** and your profile in order to be set consistently. These options are adjusted with the **set** built-in:

```
$ set -x
```

The **-x** option causes bash to print out each command it is about to run:

```
$ echo $FOO
+ echo foo
foo
```

This can be very useful for understanding unexpected quoting behavior or similar strangeness. To turn off the **-x** option, do **set +x**. See the **bash** man page for all of the options to the **set** built-in.

Setting variables with "set"

The **set** built-in can also be used for setting variables, but when used that way it is optional. The bash command **set FOO=foo** means exactly the same as **FOO=foo**. Un-setting a variable is done with the **unset** built-in:

```
$ FOO=bar
$ echo $FOO
bar
$ unset FOO
$ echo $FOO
```

Unset vs. FOO=

This is not the same as setting a variable to nothing, although it is sometimes hard to tell the difference. One way to tell is to use the **set** built-in with no parameters to list all current variables:

```
$ FOO=bar
$ set | grep ^FOO
FOO=bar
$ FOO=
$ set | grep ^FOO
FOO=
$ unset FOO
$ set | grep ^FOO
```

Using **set** with no parameters like this is similar to using the **export** built-in, except that **set** lists all variables instead of just those marked for export.

Exporting to change command behavior

Often, the behavior of commands can be altered by setting environment variables. Just as with new bash sessions, other programs that are started from your bash prompt will only be able to see variables that are marked for export. For example, the command **man** checks the variable **PAGER** to see what program to use to step through the text one page at a time.

```
$ PAGER=less
$ export PAGER
$ man man
```

With **PAGER** set to **less**, you will see one page at a time, and pressing the space bar moves on to the next page. If you change **PAGER** to **cat**, the text will be displayed all at once, without stopping.

```
$ PAGER=cat
```

```
$ man man
```

Using "env"

Unfortunately, if you forget to set **PAGER** back to **less**, **man** (as well as some other commands) will continue to display all their text without stopping. If you wanted to have **PAGER** set to **cat** just once, you could use the **env** command:

```
$ PAGER=less
$ env PAGER=cat man man
$ echo $PAGER
less
```

This time, **PAGER** was exported to **man** with a value of **cat**, but the **PAGER** variable itself remained unchanged in the bash session.

Section 6. Resources and feedback

Until next time...

In the meantime, be sure to check out the various Linux documentation resources covered in this tutorial -- particularly *http://www.linuxdoc.org*. You'll find linuxdoc's collection of guides, HOWTOs, FAQs and man pages to be invaluable. Be sure to check out *Linux Gazette* and *LinuxFocus* as well.

The Linux System Administrators guide, available from *Linuxdoc.org's "Guides" section*, is a good complement to this series of tutorials -- give it a read! You may also find Eric S. Raymond's *Unix and Internet Fundamentals HOWTO* to be helpful.

You can read the GNU Project's online documentation for the GNU info system (also called "texinfo") at *GNU's texinfo documentation page*.

In the *Bash by example* article series on *developerWorks*, Daniel shows you how to use **bash** programming constructs to write your own **bash** scripts. This bash series (particularly Parts 1 and 2) will be good preparation for the LPIC Level 1 exam and will help reinforce the concepts covered in this tutorial's "Tuning the user environment" section:

* *Bash by example, Part 1: Fundamental programming in the Bourne-again shell*
* *Bash by example, Part 2: More bash programming fundamentals*
* *Bash by example, Part 3: Exploring the ebuild system*

We highly recommend the *Technical FAQ by Linux Users* by Mark Chapman, a 50-page in-depth list of frequently-asked Linux questions, along with detailed answers. The FAQ itself is in PDF (Adobe Acrobat) format. If you're a beginning or intermediate Linux user, you really owe it to yourself to check this FAQ out. We also recommend the *Linux glossary for Linux users*, also from Mark.

If you're not familiar with the **vi** editor, we strongly recommend that you check out Daniel's *Vi intro -- the cheat sheet method tutorial* . This tutorial will give you a gentle yet fast-paced introduction to this powerful text editor. Consider this must-read material if you don't know how to use **vi**.

Your feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact Daniel Robbins directly at *drobbins@gentoo.org*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats

from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.