LPI certification 101 exam prep, Part 1

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 "Linux fundamentals", the first of four tutorials designed to prepare you for the Linux Professional Institute's 101 exam. In this tutorial, we'll introduce you to **bash** (the standard Linux shell), show you how to take full advantage of standard Linux commands like **ls**, **cp**, and **mv**, explain Linux's permission and ownership model, and much more.

By the end of this tutorial, you'll have a solid grounding in Linux fundamentals and will even be ready to begin learning some basic Linux system administration tasks.

By the end of this *series* of tutorials (eight in all), you'll 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.

Should I take this tutorial?

This tutorial (Part 1) is ideal for those who are new to Linux, or those who want to review or improve their understanding of fundamental Linux concepts, such as copying files, moving files, and creating symbolic and hard links. Along the way, we'll share plenty of hints, tips, and tricks to keep the tutorial "meaty" and practical, even for those with a good amount of previous Linux experience. For beginners, 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 fundamental Linux skills.

Also in this series are three other tutorials:

* *Part 2: Basic administration*
* *Part 3: Intermediate administration*
* *Part 4: Advanced administration*

About the author

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org*.

Residing in Albuquerque, New Mexico, Daniel Robbins 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 his new baby daughter, Hadassah.

Section 2. Introducing bash

The shell

If you've used a Linux system, you know that when you log in, you are greeted by a prompt that looks something like this:

```
$
```

The particular prompt that you see may look quite different. It may contain your system's hostname, the name of the current working directory, or both. But regardless of what your particular prompt may look like, there's one thing that's certain. The program that printed that prompt is called a "shell", and it's very likely that your particular shell is a program called **bash**.

Are you running bash?

You can check to see if you're running **bash** by typing:

```
$ echo $SHELL
/bin/bash
```

If the above line gave you an error or didn't respond similarly to our example, then you may be running a shell other than **bash**. In that case, most of this tutorial should still apply, but it would be advantageous for you to switch to **bash** for the sake of preparing for the 101 exam. (See Part 2 of this tutorial series for information on changing your shell using the **chsh** command.)

About bash

Bash, an acronym for "Bourne-again shell", is the default shell on most Linux systems. The shell's job is to obey your commands so that you can interact with your Linux system. When you're finished entering commands, you may instruct the shell to **exit** or **logout**, at which point you'll be returned to a login prompt.

By the way, you can also logout by pressing control-D at the **bash** prompt.

Using "cd"

As you've probably found, staring at your **bash** prompt isn't the most exciting thing in the world. So, let's start using **bash** to navigate around our filesystem. At the prompt, type the following (without the **$**):

```
$ cd /
```

We've just told **bash** that you want to work in /, also known as the *root* directory; all the directories on the system form a tree, and / is considered the top of this tree, or the root. **cd**

sets the directory where you are currently working, also known as the "current working directory".

Paths

To see bash's current working directory, you can type:

```
$ pwd
/
```

In the above example, the / argument to **cd** is called a *path*. It tells **cd** where we want to go. In particular, the / argument is an *absolute* path, meaning that it specifies a location relative to the root of the filesystem tree.

Absolute paths

Here are some other absolute paths:

```
/dev
/usr
/usr/bin
/usr/local/bin
```

As you can see, the one thing that all absolute paths have in common is that they begin with /. With a path of /usr/local/bin, we're telling **cd** to enter the / directory, then the usr directory under that, and then local and bin. Absolute paths are always evaluated by starting at / first.

Relative paths

The other kind of path is called a *relative path*. Bash, **cd**, and other commands always interpret these paths relative to the current directory. Relative paths never begin with a /. So, if we're in /usr:

```
$ cd /usr
```

Then, we can use a relative path to change to the /usr/local/bin directory:

```
$ cd local/bin
$ pwd
/usr/local/bin
```

Using ".."

Relative paths may also contain one or more .. directories. The .. directory is a special directory that points to the parent directory. So, continuing from the example above:

```
$ pwd
```

```
/usr/local/bin
$ cd ..
$ pwd
/usr/local
```

As you can see, our current directory is now /usr/local. We were able to go "backwards" one directory, relative to the current directory that we were in.

Using "..", continued

In addition, we can also add .. to an existing relative path, allowing us to go into a directory that's alongside one we are already in, for example:

```
$ pwd
/usr/local
$ cd ../share
$ pwd
/usr/share
```

Relative path examples

Relative paths can get quite complex. Here are a few examples, all without the resultant target directory displayed. Try to figure out where you'll end up after typing these commands:

```
$ cd /bin
$ cd ../usr/share/zoneinfo
```

```
$ cd /usr/X11R6/bin
$ cd ../lib/X11
```

```
$ cd /usr/bin
$ cd ../bin/../bin
```

Now, try them out and see if you got them right. :)

Understanding "."

Before we finish our coverage of **cd**, we need to discuss a few more things. First, there is another special directory called **.**, which means "the current directory". While this directory isn't used with the **cd** command, it's often used to execute some program in the current directory, as follows:

```
$ ./myprog
```

In the above example, the **myprog** executable residing in the current working directory will be executed.

cd and the home directory

If we wanted to change to our home directory, we could type:

```
$ cd
```

With no arguments, cd will change to your home directory, which is /root for the superuser and typically /home/username for a regular user. But what if we want to specify a file in our home directory? Maybe we want to pass a file argument to the **myprog** command. If the file lives in our home directory, we can type:

```
$ ./myprog /home/drobbins/myfile.txt
```

However, using an absolute path like that isn't always convenient. Thankfully, we can use the **~** (tilde) character to do the same thing:

```
$ ./myprog ~/myfile.txt
```

Other users' home directories

Bash will expand a lone **~** to point to your home directory, but you can also use it to point to other users' home directories. For example, if we wanted to refer to a file called fredsfile.txt in fred's home directory, we could type:

```
$ ./myprog ~fred/fredsfile.txt
```

Section 3. Using Linux commands

Introducing "ls"

Now, we'll take a quick look at the **ls** command. Very likely, you're already familiar with **ls** and know that typing it by itself will list the contents of the current working directory:

```
$ cd /usr
$ ls
X11R6      doc         i686-pc-linux-gnu  lib      man          sbin   ssl
bin        gentoo-x86  include            libexec  portage      share  tmp
distfiles  i686-linux  info               local    portage.old  src
```

By specifying the **-a** option, you can see all of the files in a directory, including hidden files -- those that begin with **.**. As you can see in the following example, **ls -a** reveals the **.** and **..** special directory links:

```
$ ls -a
.      bin        gentoo-x86         include  libexec  portage      share  tmp
..     distfiles  i686-linux         info     local    portage.old  src
X11R6  doc        i686-pc-linux-gnu  lib      man      sbin         ssl
```

Long directory listings

You can also specify one or more files or directories on the **ls** command line. If you specify a file, **ls** will show that file only. If you specify a directory, **ls** will show the *contents* of the directory. The **-l** option comes in very handy when you need to view permissions, ownership, modification time, and size information in your directory listing.

Long directory listings, continued

In the following example, we use the **-l** option to display a full listing of my /usr directory.

```
$ ls -l /usr
drwxr-xr-x    7 root     root          168 Nov 24 14:02 X11R6
drwxr-xr-x    2 root     root        14576 Dec 27 08:56 bin
drwxr-xr-x    2 root     root         8856 Dec 26 12:47 distfiles
lrwxrwxrwx    1 root     root            9 Dec 22 20:57 doc -> share/doc
drwxr-xr-x   62 root     root         1856 Dec 27 15:54 gentoo-x86
drwxr-xr-x    4 root     root          152 Dec 12 23:10 i686-linux
drwxr-xr-x    4 root     root           96 Nov 24 13:17 i686-pc-linux-gnu
drwxr-xr-x   54 root     root         5992 Dec 24 22:30 include
lrwxrwxrwx    1 root     root           10 Dec 22 20:57 info -> share/info
drwxr-xr-x   28 root     root        13552 Dec 26 00:31 lib
drwxr-xr-x    3 root     root           72 Nov 25 00:34 libexec
drwxr-xr-x    8 root     root          240 Dec 22 20:57 local
lrwxrwxrwx    1 root     root            9 Dec 22 20:57 man -> share/man
lrwxrwxrwx    1 root     root           11 Dec  8 07:59 portage -> gentoo-x86/
drwxr-xr-x   60 root     root         1864 Dec  8 07:55 portage.old
drwxr-xr-x    3 root     root         3096 Dec 22 20:57 sbin
drwxr-xr-x   46 root     root         1144 Dec 24 15:32 share
drwxr-xr-x    8 root     root          328 Dec 26 00:07 src
drwxr-xr-x    6 root     root          176 Nov 24 14:25 ssl
```

```
lrwxrwxrwx    1 root     root           10 Dec 22 20:57 tmp -> ../var/tmp
```

The first column displays permissions information for each item in the listing. I'll explain how to interpret this information in a bit. The next column lists the number of links to each filesystem object, which we'll gloss over now but return to later. The third and fourth columns list the owner and group, respectively. The fifth column lists the object size. The sixth column is the "last modified" time or "mtime" of the object. The last column is the object's name. If the file is a symbolic link, you'll see a trailing **->** and the path to which the symbolic link points.

Looking at directories

Sometimes, you'll want to look *at* a directory, rather than inside it. For these situations, you can specify the **-d** option, which will tell **ls** to look at any directories that it would normally look inside:

```
$ ls -dl /usr /usr/bin /usr/X11R6/bin ../share
drwxr-xr-x    4 root     root           96 Dec 18 18:17 ../share
drwxr-xr-x   17 root     root          576 Dec 24 09:03 /usr
drwxr-xr-x    2 root     root         3192 Dec 26 12:52 /usr/X11R6/bin
drwxr-xr-x    2 root     root        14576 Dec 27 08:56 /usr/bin
```

Recursive and inode listings

So you can use **-d** to look *at* a directory, but you can also use **-R** to do the opposite -- not just look inside a directory, but recursively look inside all the directories inside that directory! We won't include any example output for this option (since it's generally voluminous), but you may want to try a few **ls -R** and **ls -Rl** commands to get a feel for how this works.

Finally, the **-i** ls option can be used to display the *inode numbers* of the filesystem objects in the listing:

```
$ ls -i /usr
   1409 X11R6        314258 i686-linux           43090 libexec        13394 sbin
   1417 bin            1513 i686-pc-linux-gnu     5120 local          13408 share
   8316 distfiles      1517 include                776 man            23779 src
     43 doc            1386 info                 93892 portage        36737 ssl
  70744 gentoo-x86     1585 lib                   5132 portage.old       784 tmp
```

Understanding inodes, part 1

Every object on a filesystem is assigned a unique index, called an inode number. This might seem trivial, but understanding inodes is essential to understanding many filesystem operations. For example, consider the **.** and **..** links that appear in every directory. To fully understand what a **..** directory actually is, we'll first take a look at /usr/local's inode number:

```
$ ls -id /usr/local
   5120 /usr/local
```

The /usr/local directory has an inode number of 5120. Now, let's take a look at the inode number of /usr/local/bin/..:

```
$ ls -id /usr/local/bin/..
   5120 /usr/local/bin/..
```

Understanding inodes, part 2

As you can see, /usr/local/bin/.. has the same inode number as /usr/local! Here's how can we come to grips with this shocking revelation. In the past, we've considered /usr/local to be *the* directory itself. Now, we discover that inode 5120 is in fact *the* directory, and we have found two directory entries (called "links") that point to this inode. Both /usr/local and /usr/local/bin/.. are *links* to inode 5120. Although inode 5120 exists in only one place on disk, multiple things link to it.

Understanding inodes, part 3

In fact, we can see the total number of times that inode 5120 is referenced by using the **ls -dl** command:

```
$ ls -dl /usr/local
drwxr-xr-x    8 root     root          240 Dec 22 20:57 /usr/local
```

If we take a look at the second column from the left, we see that the directory /usr/local (inode 5120) is referenced eight times. On my system, here are the various paths that reference this inode:

```
/usr/local
/usr/local/.
/usr/local/bin/..
/usr/local/games/..
/usr/local/lib/..
/usr/local/sbin/..
/usr/local/share/..
/usr/local/src/..
```

mkdir

Let's take a quick look at the **mkdir** command, which can be used to create new directories. The following example creates three new directories, tic, tac, and toe, all under /tmp:

```
$ cd /tmp
$ mkdir tic tac toe
```

By default, the **mkdir** command doesn't create parent directories for you; the entire path up to the next-to-last element needs to exist. So, if you want to create the directories won/der/ful, you'd need to issue three separate **mkdir** commands:

```
$ mkdir won/der/ful
```

```
mkdir: cannot create directory `won/der/ful': No such file or directory
$ mkdir won
$ mkdir won/der
$ mkdir won/der/ful
```

mkdir -p

However, mkdir has a handy **-p** option that tells mkdir to create any missing parent directories, as can be seen here:

```
$ mkdir -p easy/as/pie
```

All in all, pretty straightforward. To learn more about the **mkdir** command, type **man mkdir** to read the manual page. This will work for nearly all commands covered here (such as **man ls**), except for **cd**, which is built-in to **bash**.

touch

Now, we're going to take a quick look at the **cp** and **mv** commands, used to copy, rename, and move files and directories. To begin this overview, we'll first use the **touch** command to create a file in /tmp:

```
$ cd /tmp
$ touch copyme
```

The **touch** command updates the "mtime" of a file if it exists (recall the sixth column in **ls -l** output). If the file doesn't exist, then a new, empty file will be created. You should now have a /tmp/copyme file with a size of zero.

echo and redirection

Now that the file exists, let's add some data to the file. We can do this using the **echo** command, which takes its arguments and prints them to standard output. First, the echo command by itself:

```
$ echo "firstfile"
firstfile
```

echo and redirection

Now, the same **echo** command with output redirection:

```
$ echo "firstfile" > copyme
```

The greater-than sign tells the shell to write **echo**'s output to a file called copyme. This file will be created if it doesn't exist, and will be overwritten if it does exist. By typing **ls -l**, we can

see that the copyme file is 10 bytes long, since it contains the word **firstfile** and the newline character:

```
$ ls -l copyme
-rw-r--r--    1 root     root           10 Dec 28 14:13 copyme
```

cat and cp

To display the contents of the file on the terminal, use the **cat** command:

```
$ cat copyme
firstfile
```

Now, we can use a basic invocation of the **cp** command to create a copiedme file from the original copyme file:

```
$ cp copyme copiedme
```

Upon investigation, we find that they are truly separate files; their inode numbers are different:

```
$ ls -i copyme copiedme
  648284 copiedme   650704 copyme
```

mv

Now, let's use the **mv** command to rename "copiedme" to "movedme". The inode number will remain the same; however, the filename that points to the inode will change.

```
$ mv copiedme movedme
$ ls -i movedme
  648284 movedme
```

A moved file's inode number will remain the same as long as the destination file resides on the same filesystem as the source file. We'll take a closer look at filesystems in Part 3 of this tutorial series.

Section 4. Creating links and removing files

Hard links

We've mentioned the term *link* when referring to the relationship between directory entries and inodes. There are actually two kinds of links available on Linux. The kind we've discussed so far are called *hard links*. A given inode can have any number of hard links, and the inode will persist on the filesystem until all the hard links disappear. New hard links can be created using the **ln** command:

```
$ cd /tmp
$ touch firstlink
$ ln firstlink secondlink
$ ls -i firstlink secondlink
  15782 firstlink    15782 secondlink
```

Hard links, continued

As you can see, hard links work on the inode level to point to a particular file. On Linux systems, hard links have several limitations. For one, you can only make hard links to files, not directories. That's right; even though **.** and **..** are system-created hard links to directories, you (not even as the "root" user) aren't allowed to create any of your own.

The second limitation of hard links is that they can't span filesystems. This means that you can't create a link from /usr/bin/bash to /bin/bash if your / and /usr directories exist on separate filesystems.

Symbolic links

In practice, symbolic links (or "symlinks") are used more often than hard links. Symlinks are a special file type where the link refers to another file by name, rather than directly to the inode. Symlinks do not prevent a file from being deleted; if the target file disappears, then the symlink will just be unusable, or "broken".

Symbolic links, continued

A symbolic link can be created by passing the **-s** option to **ln**.

```
$ ln -s secondlink thirdlink
$ ls -l firstlink secondlink thirdlink
-rw-rw-r--    2 agriffis agriffis        0 Dec 31 19:08 firstlink
-rw-rw-r--    2 agriffis agriffis        0 Dec 31 19:08 secondlink
lrwxrwxrwx    1 agriffis agriffis       10 Dec 31 19:39 thirdlink -> secondlink
```

Symbolic links can be distinguished in **ls -l** output from normal files in three ways. First, notice that the first column contains an **l** character to signify the symbolic link. Second, the size of the symbolic link is the number of characters in the target (**secondlink** in this case). Third, the last column of the output displays the target filename.

Symlinks in-depth, part 1

Symbolic links are generally more flexible than hard links. You can create a symbolic link to any type of filesystem object, including directories. And because the implementation of symbolic links is based on paths (not inodes), it's perfectly fine to create a symbolic link that points to an object on another filesystem. However, this fact can also make symbolic links tricky to understand.

Symlinks in-depth, part 2

Consider a situation where we want to create a link in /tmp that points to /usr/local/bin. Should we type this:

```
$ ln -s /usr/local/bin bin1
$ ls -l bin1
lrwxrwxrwx    1 root     root           14 Jan  1 15:42 bin1 -> /usr/local/bin
```

Or alternatively:

```
$ ln -s ../usr/local/bin bin2
$ ls -l bin2
lrwxrwxrwx    1 root     root           16 Jan  1 15:43 bin2 -> ../usr/local/bin
```

Symlinks in-depth, part 3

As you can see, both symbolic links point to the same directory. However, if our second symbolic link is ever moved to another directory, it will be "broken" because of the relative path:

```
$ ls -l bin2
lrwxrwxrwx    1 root     root           16 Jan  1 15:43 bin2 -> ../usr/local/bin
$ mkdir mynewdir
$ mv bin2 mynewdir
$ cd mynewdir
$ cd bin2
bash: cd: bin2: No such file or directory
```

Because the directory /tmp/usr/local/bin doesn't exist, we can no longer change directories into bin2; in other words, bin2 is now broken.

Symlinks in-depth, part 4

For this reason, it is sometimes a good idea to avoid creating symbolic links with relative path information. However, there are many cases where relative symbolic links come in handy. Consider an example where you want to create an alternate name for a program in /usr/bin:

```
# ls -l /usr/bin/keychain
-rwxr-xr-x    1 root     root        10150 Dec 12 20:09 /usr/bin/keychain
```

Symlinks in-depth, part 5

As the root user, you may want to create an alternate name for "keychain", such as "kc". In this example, we have root access, as evidenced by our bash prompt changing to "#". We need root access because normal users aren't able to create files in /usr/bin. As root, we could create an alternate name for keychain as follows:

```
# cd /usr/bin
# ln -s /usr/bin/keychain kc
```

Symlinks in-depth, part 6

While this solution will work, it will create problems if we decide that we want to move both files to /usr/local/bin:

```
# mv /usr/bin/keychain /usr/bin/kc /usr/local/bin
```

Because we used an *absolute* path in our symbolic link, our **kc** symlink is still pointing to /usr/bin/keychain, which no longer exists -- another broken symlink. Both relative and absolute paths in symbolic links have their merits, and you should use a type of path that's appropriate for your particular application. Often, either a relative or absolute path will work just fine. In this case, the following example would have worked:

```
# cd /usr/bin
# ln -s keychain kc
# ls -l kc
lrwxrwxrwx    1 root     root            8 Jan  5 12:40 kc -> keychain
```

rm

Now that we know how to use **cp**, **mv**, and **ln**, it's time to learn how to remove objects from the filesystem. Normally, this is done with the **rm** command. To remove files, simply specify them on the command line:

```
$ cd /tmp
$ touch file1 file2
$ ls -l file1 file2
-rw-r--r--    1 root     root            0 Jan  1 16:41 file1
-rw-r--r--    1 root     root            0 Jan  1 16:41 file2
$ rm file1 file2
$ ls -l file1 file2
ls: file1: No such file or directory
ls: file2: No such file or directory
```

rmdir

To remove directories, you have two options. You can remove all the objects inside the directory and then use **rmdir** to remove the directory itself:

```
$ mkdir mydir
$ touch mydir/file1
$ rm mydir/file1
$ rmdir mydir
```

rm and directories

Or, you can use the *recursive force* options of the **rm** command to tell rm to remove the directory you specify, as well as all objects contained in the directory:

```
$ rm -rf mydir
```

Generally, **rm -rf** is the preferred method of removing a directory tree. Be very careful when using **rm -rf**, since its power can be used for both good and evil. :)

Section 5. Introducing wildcards

Introducing wildcards

In your day-to-day Linux use, there are many times when you may need to perform a single operation (such as **rm**) on many filesystem objects at once. In these situations, it can often be cumbersome to type in many files on the command line:

```
$ rm file1 file2 file3 file4 file5 file6 file7 file8
```

Introducing wildcards, continued

To solve this problem, you can take advantage of Linux's built-in wildcard support. This support, also called "globbing" (for historical reasons), allows you to specify multiple files at once by using a *wildcard pattern*. Bash and other Linux commands will interpret this pattern by looking on disk and finding any files that match it. So, if you had files file1 through file8 in the current working directory, you could remove these files by typing:

```
$ rm file[1-8]
```

Or if you simply wanted to remove all files whose names begin with file, you could type:

```
$ rm file*
```

Understanding non-matches

Or if you wanted to list all the filesystem objects in /etc beginning with **g**, you could type:

```
$ ls -d /etc/g*
/etc/gconf  /etc/ggi  /etc/gimp  /etc/gnome  /etc/gnome-vfs-mime-magic  /etc/gpm  /etc/
```

Now, what happens if you specify a pattern that doesn't match any filesystem objects? In the following example, we try to list all the files in /usr/bin that begin with **asdf** and end with **jkl**:

```
$ ls -d /usr/bin/asdf*jkl
ls: /usr/bin/asdf*jkl: No such file or directory
```

Understanding non-matches, continued

Here's what happened. Normally, when we specify a pattern, that pattern matches one or more files on the underlying filesystem, and **bash** *replaces the pattern with a space-separated list of all matching objects* . However, when the pattern doesn't produce any matches, **bash** *leaves the argument, wildcards and all, as is.* So when "ls" can't find the file /usr/bin/asdf*jkl, it gives an error. The operative rule here is that *glob patterns are expanded only if they match objects in the filesystem*.

Wildcard syntax: *

Now that we understand *how* globbing works, let's review wildcard syntax. You can use several special characters for wildcard expansion; here's one:

```
*
```

***** will match zero or more characters. It means "anything can go here". Examples:

* **/etc/g*** matches all files in /etc that begin with **g**.
* **/tmp/my*1** matches all files in /tmp that begin with **my** and end with **1**.

Wildcard syntax: ?

```
?
```

? matches any single character. Examples:

* **myfile?** matches any file whose name consists of **myfile** followed by a single character.
* **/tmp/notes?txt** would match both **/tmp/notes.txt** and **/tmp/notes_txt**, if they exist.

Wildcard syntax: []

```
[]
```

This wildcard is like a **?**, but allows more specificity. To use this wildcard, place any characters you'd like to match inside the **[]**. The resultant expression will match a single occurrence of any of these characters. You can also use **-** to specify a range, and even combine ranges. Examples:

* **myfile[12]** will match **myfile1** and **myfile2**. The wildcard will be expanded as long as at least one of these files exists in the current directory.
* **[Cc]hange[Ll]og** will match **Changelog**, **ChangeLog**, **changeLog**, and **changelog**. As you can see, using bracket wildcards can be useful for matching variations in capitalization.
* **ls /etc/[0-9]*** will list all files in /etc that begin with a number.
* **ls /tmp/[A-Za-z]*** will list all files in /tmp that begin with an upper or lower-case letter.

Wildcard syntax: [!]

```
[!]
```

The **[!]** construct is similar to the **[]** construct, except rather than matching any characters inside the brackets, it'll match any character, as long as it is *not* listed between the **[!** and **]**. Examples:

* **rm myfile[!9]** will remove all files named **myfile** plus a single character, except for **myfile9**.

Wildcard caveats

Here are some caveats to watch out for when using wildcards. Since **bash** treats wildcard-related characters (**?**, **[**, **]**, *****) specially, you need to take special care when typing in an argument to a command that contains these characters. For example, if you want to create a file that contains the string **[fo]***, the following command may not do what you want:

```
$ echo [fo]* > /tmp/mynewfile.txt
```

If the pattern **[fo]*** matches any files in the current working directory, then you'll find the names of those files inside /tmp/mynewfile.txt rather than a literal **[fo]*** like you were expecting. The solution? Well, one approach is to surround your characters with single quotes, which tell **bash** to perform absolutely no wildcard expansion on them:

```
$ echo '[fo]*' > /tmp/mynewfile.txt
```

Wildcard caveats, continued

Using this approach, your new file will contain a literal **[fo]*** as expected. Alternatively, you could use backslash escaping to tell **bash** that **[**, **]** and ***** should be treated literally rather than as wildcards:

```
$ echo \[fo\]\* > /tmp/mynewfile.txt
```

Both approaches will work identically. Since we're talking about backslash expansion, now would be a good time to mention that in order to specify a literal ****, you can either enclose it in single quotes as well, or type **** instead (it will be expanded to ****).

Single vs. double quotes

Note that double quotes will work similarly to single quotes, but will still allow **bash** to do some limited expansion. Therefore, single quotes are your best bet when you are truly interested in passing literal text to a command. For more information on wildcard expansion, type **man 7 glob**. For more information on quoting in **bash**, type **man 8 glob** and read the section titled **QUOTING**. If you're planning to take the LPI exams, consider this a homework assignment. :)

Section 6. Resources and feedback

Resources and homework

Congratulations; you've reached the end of "Linux fundamentals". I hope that this tutorial has helped you to firm up your foundational Linux knowledge. Please join us in our next tutorial on "Basic administration," where we'll build on the foundation laid here, covering topics like regular expressions, ownership and permissions, user account management, and more. The other tutorials in this series are:

* *Part 2: Basic administration*
* *Part 3: Intermediate administration*
* *Part 4: Advanced administration*

And remember, by completing this tutorial series, you'll be prepared to attain your LPIC Level 1 Certification from the Linux Professional Institute. Speaking of LPIC certification, if this is something you're interested in, then we recommend that you study the following resources, which augment the material covered in this tutorial:

In the *Bash by example* article series on developerWorks, I show 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:

* *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*

I highly recommend the *Technical FAQ for Linux users*, 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.

If you're not too familiar with the **vi** editor, I strongly recommend that you check out my *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

I look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact me 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.

LPI certification 101 exam prep, Part 2

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 "Basic administration", the second of four tutorials designed to prepare you for the Linux Professional Institute's 101 exam. In this tutorial, we'll show you how to use regular expressions to search files for text patterns. Next, we'll introduce you to the Filesystem Hierarchy Standard, or FHS, and then show you how to locate files on your system. Then, we'll show you how to take full control of Linux processes by running them in the background, listing processes, detaching processes from the terminal, and more. Finally, we'll give you a whirlwind introduction to shell pipelines, redirection, and text processing commands.

By the end of this tutorial, you'll have a solid grounding in basic Linux administration and will be ready to begin learning some more advanced Linux system administration skills.

By the end of this *series* of tutorials (eight in all), you'll 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.

Should I take this tutorial?

This tutorial (Part 2) is ideal for those who have a good basic knowledge of bash, and want to receive a solid introduction to basic Linux administration tasks. If you are new to Linux, we recommend that you complete *Part 1* of this tutorial series first before continuing. For some, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way to "round out" their basic Linux administration skills.

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. Regular expressions

What is a regular expression?

A regular expression (also called a "regex" or "regexp") is a special syntax used to describe text patterns. On Linux systems, regular expressions are commonly used to find patterns of text, as well as perform search-and-replace operations on text streams, among other things.

Glob comparison

As we look at regular expressions, you may find that regular expression syntax looks similar to filename "globbing" syntax that we examined in our previous tutorial (see Part 1 listed in the Resources section at the end of this tutorial). However, don't let this fool you; their similarity is only skin-deep. Both regular expressions and filename globbing patterns, while they may look similar, are fundamentally different beasts.

The simple substring

With that caution, let's look at the most basic of regular expressions, the *simple substring*. To do this, we're going to use `grep`, a command that scans the contents of a file for a particular regular expression. `grep` prints every line that matches the regular expression, and ignores every line that doesn't:

```
$ grep bash /etc/passwd
operator:x:11:0:operator:/root:/bin/bash
root:x:0:0::/root:/bin/bash
ftp:x:40:1::/home/ftp:/bin/bash
```

Above, the first parameter to `grep` is a regex; the second is a filename. `Grep` read each line in `/etc/passwd` and applied the simple substring regex `bash` to it, looking for a match. If a match was found, `grep` printed out the entire line; otherwise, the line was ignored.

Understanding the simple substring

In general, if you are searching for a substring, you can just specify the text verbatim without supplying any "special" characters. The only time you'd need to do anything special would be if your substring contained a `+`, `.`, `*`, `[`, `]`, or `\`, in which case these characters would need enclosed in quotes and preceded by a backslash. Here are a few more examples of simple substring regular expressions:

* `/tmp` (scans for the literal string `/tmp`)
* `"\[box\]"` (scans for the literal string `[box]`)
* `"\*funny\*"` (scans for the literal string `*funny*`)
* `"ld\.so"` (scans for the literal string `ld.so`)

Metacharacters

With regular expressions, you can perform much more complex searches than the examples we've looked at so far by taking advantage of *metacharacters*. One of these metacharacters is the `.` (a period), which matches any single character:

```
$ grep dev.hda /etc/fstab
/dev/hda3       /               reiserfs        noatime,ro 1 1
/dev/hda1       /boot           reiserfs        noauto,noatime,notail 1 2
/dev/hda2       swap            swap            sw 0 0
#/dev/hda4      /mnt/extra      reiserfs        noatime,rw 1 1
```

In this example, the literal text `dev.hda` didn't appear on any of the lines in `/etc/fstab`. However, grep wasn't scanning them for the literal `dev.hda` string, but for the `dev.hda` *pattern*. Remember that the `.` will match *any single character*. As you can see, the `.` metacharacter is functionally equivalent to how the ? metacharacter works in glob expansions.

Using []

If we wanted to match a character a bit more specifically than `.`, we could use `[` and `]` (square brackets) to specify a subset of characters that should be matched:

```
$ grep dev.hda[12] /etc/fstab
/dev/hda1       /boot           reiserfs        noauto,noatime,notail 1 2
/dev/hda2       swap            swap            sw 0 0
```

As you can see, this particular syntactical feature works identically to the `[]` in "glob" filename expansions. Again, this is one of the tricky things about learning regular expressions -- the syntax is *similar* but not identical to "glob" filename expansion syntax, which often makes regexes a bit confusing to learn.

Using [^]

You can reverse the meaning of the square brackets by putting a `^` immediately after the `[`. In this case, the brackets will match any character that is not listed inside the brackets. Again, note that we use `[^]` with regular expressions, but `[!]` with globs:

```
$ grep dev.hda[^12] /etc/fstab
/dev/hda3       /               reiserfs        noatime,ro 1 1
#/dev/hda4      /mnt/extra      reiserfs        noatime,rw 1 1
```

Differing syntax

It's important to note that the syntax *inside* square brackets is fundamentally different from that in other parts of the regular expression. For example, if you put a `.` inside square brackets, it allows the square brackets to match a literal `.`, just like the `1` and `2` in the examples above. In comparison, a literal `.` outside the square brackets is interpreted as a metacharacter unless prefixed by a `\`. We can take advantage of this fact to print a list of all lines in `/etc/fstab` that contain the literal string `dev.hda` by typing:

```
$ grep dev[.]hda /etc/fstab
```

Alternately, we could also type:

```
$ grep "dev\.hda" /etc/fstab
```

Neither regular expressions are likely to match any lines in your /etc/fstab file.

The "*" metacharacter

Some metacharacters don't match anything in themselves, but instead modify the meaning of a previous character. One such metacharacter is * (asterisk), which is used to match zero or more repeated occurrences of the previous character. Here are some examples:

* ab*c (matches abbbbc but not abqc)
* ab*c (matches abc but not abbqbbc)
* ab*c (matches ac but not cba)
* b[cq]*e (matches bqe but not eb)
* b[cq]*e (matches bccqqe but not bccc)
* b[cq]*e (matches bqqcce but not cqe)
* b[cq]*e (matches bbbeee)
* .* (matches any string)
* foo.* (matches any string that begins with foo)

The line ac matches the regex ab*c because the asterisk also allows the preceding expression (c) to appear zero times. Note that the * regex metacharacter is interpreted in a fundamentally different way than the * glob character.

Beginning and end of line

The last metacharacters we will cover in detail here are the ^ and $ metacharacters, used to match the beginning and end of line, respectively. By using a ^ at the beginning of your regex, you can cause your pattern to be "anchored" to the start of the line. In the following example, we use the ^# regex to match any line beginning with the # character:

```
$ grep ^# /etc/fstab
# /etc/fstab: static file system information.
#
```

Full line regexps

^ and $ can be combined to match an entire line. For example, the following regex will match a line that starts with the # character and ends with the . character, with any number of other characters in-between:

```
$ grep '^#.*\.$' /etc/fstab
# /etc/fstab: static file system information.
```

In the above example, we surrounded our regular expression with single quotes to prevent $ from being interpreted by the shell. Without the single quotes, the $ will disappear from our regex before grep even has a chance to take a look at it.

Section 3. FHS and finding files

Filesystem Hierarchy Standard

The *Filesystem Hierarchy Standard* is a document that specifies the layout of directories on a Linux system. The FHS was devised to provide a common layout to simplify distribution-independent software development. The FHS specifies the following directories (taken directly from the FHS specification):

* `/` (the root directory)
* `/boot` (static files of the boot loader)
* `/dev` (device files)
* `/etc` (host-specific system configuration)
* `/lib` (essential shared libraries and kernel modules)
* `/mnt` (mount point for mounting a filesystem temporarily)
* `/opt` (add-on application software packages)
* `/sbin` (essential system binaries)
* `/tmp` (temporary files)
* `/usr` (secondary hierarchy)
* `/var` (variable data)

The two independent FHS categories

The FHS bases its layout specification on the idea that there are two independent categories of files: shareable vs. unshareable, and variable vs. static. *Shareable data* can be shared between hosts; *unshareable data* is specific to a given host (such as configuration files). *Variable data* can be modified; *static data* is not modified (except at system installation and maintenance).

The following grid summarizes the four possible combinations, with examples of directories that would fall into those categories. Again, this table is straight from the FHS specification:

```
+---------+-----------------+-------------+
|         | shareable       | unshareable |
+---------+-----------------+-------------+
|static   | /usr            | /etc        |
|         | /opt            | /boot       |
+---------+-----------------+-------------+
|variable | /var/mail       | /var/run    |
|         | /var/spool/news | /var/lock   |
+---------+-----------------+-------------+
```

Secondary hierarchy at /usr

Under `/usr` you'll find a secondary hierarchy that looks a lot like the root filesystem. It isn't critical for `/usr` to exist when the machine powers up, so it can be shared on a network ("shareable"), or mounted from a CD-ROM ("static"). Most Linux setups don't make use of sharing `/usr`, but it's valuable to understand the usefulness of distinguishing between the primary hierarchy at the root directory and the secondary hierarchy at `/usr`.

This is all we'll say about the *Filesystem Hierarchy Standard*. The document itself is quite readable, so you should go take a look at it. We promise you'll understand a lot more about the Linux filesystem if you read it.

Finding files

Linux systems often contain hundreds of thousands of files. Perhaps you are savvy enough to never lose track of any of them, but it's more likely that you will occasionally need help finding one. There are a few different tools on Linux for finding files. The following panels will introduce you to them, and help you to choose the right tool for the job.

The PATH

When you run a program at the command line, bash actually searches through a list of directories to find the program you requested. For example, when you type `ls`, bash doesn't intrinsically know that the `ls` program lives in `/usr/bin`. Instead, bash refers to an environment variable called `PATH`, which is a colon-separated list of directories. We can examine the value of `PATH`:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin
```

Given this value of `PATH` (yours may differ), bash would first check `/usr/local/bin`, then `/usr/bin` for the `ls` program. Most likely, `ls` is kept in `/usr/bin`, so bash would stop at that point.

Modifying PATH

You can augment your `PATH` by assigning elements to it on the command line:

```
$ PATH=$PATH:~/bin
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin:/home/agriffis/bin
```

You can also remove elements from `PATH`, although it isn't as easy since you can't refer to the existing `$PATH`. Your best bet is to simply type out the new `PATH` you want:

```
$ PATH=/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:~/bin
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/agriffis/bin
```

All about "which"

You can check to see if there's a given program in your `PATH` by using `which`. For example, here we find out that our Linux system has no (common) sense:

```
$ which sense
```

```
which: no sense in (/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/X11R6/bin)
```

In this example, we successfully locate `ls`:

```
$ which ls
/usr/bin/ls
```

"which -a"

Finally, you should be aware of the `-a` flag, which causes `which` to show you all of the instances of a given program in your PATH:

```
$ which -a ls
/usr/bin/ls
/bin/ls
```

whereis

If you're interested in finding more information than purely the location of a program, you might try the `whereis` program:

```
$ whereis ls
ls: /bin/ls /usr/bin/ls /usr/share/man/man1/ls.1.gz
```

Here we see that `ls` occurs in two common binary locations, `/bin` and `/usr/bin`. Additionally, we are informed that there is a manual page located in `/usr/share/man`. This is the man page you would see if you were to type `man ls`.

The `whereis` program also has the ability to search for sources, to specify alternate search paths, and to search for unusual entries. Refer to the `whereis` man page for further information.

find

The `find` command is another tool for your toolbox. With `find` you aren't restricted to programs; you can search for any file you want, using a variety of search criteria. For example, to search for a file by the name of `README`, starting in `/usr/share/doc`:

```
$ find /usr/share/doc -name README
/usr/share/doc/ion-20010523/README
/usr/share/doc/bind-9.1.3-r6/dhcp-dynamic-dns-examples/README
/usr/share/doc/sane-1.0.5/README
```

find and wildcards

You can use "glob" wildcards in the argument to `-name`, provided that you quote them or

backslash-escape them (so they get passed to `find` intact rather than being expanded by `bash`) For example, we might want to search for README files with extensions:

```
$ find /usr/share/doc -name README\*
/usr/share/doc/iproute2-2.4.7/README.gz
/usr/share/doc/iproute2-2.4.7/README.iproute2+tc.gz
/usr/share/doc/iproute2-2.4.7/README.decnet.gz
/usr/share/doc/iproute2-2.4.7/examples/diffserv/README.gz
/usr/share/doc/pilot-link-0.9.6-r2/README.gz
/usr/share/doc/gnome-pilot-conduits-0.8/README.gz
/usr/share/doc/gimp-1.2.2/README.i18n.gz
/usr/share/doc/gimp-1.2.2/README.win32.gz
/usr/share/doc/gimp-1.2.2/README.gz
/usr/share/doc/gimp-1.2.2/README.perl.gz
[578 additional lines snipped]
```

Ignoring case with find

Of course, you might want to ignore case in your search:

```
$ find /usr/share/doc -name '[Rr][Ee][Aa][Dd][Mm][Ee]*'
```

Or, more simply:

```
$ find /usr/share/doc -iname readme\*
```

As you can see, you can use `-iname` to do case-insensitive searching.

find and regular expressions

If you're familiar with regular expressions, you can use the `-regex` option to limit the output to filenames that match a pattern. And similar to the `-iname` option, there is a corresponding `-iregex` option that ignores case in the pattern. For example:

Note that unlike many programs, `find` requires that the regex specified matches the entire path, not just a part of it. For that reason, specifying the leading and trailing `.*` is necessary; purely using `xt` would not be sufficient.

find and types

The `-type` option allows you to find filesystem objects of a certain type. The possible arguments to `-type` are `b` (block device), `c` (character device), `d` (directory), `p` (named pipe), `f` (regular file), `l` (symbolic link), and `s` (socket). For example, to search for symbolic links in `/usr/bin` that contain the string `vim`:

```
$ find /usr/bin -name '*vim*' -type l
/usr/bin/rvim
/usr/bin/vimdiff
/usr/bin/gvimdiff
```

find and mtimes

The `-mtime` option allows you to select files based on their last modification time. The argument to mtime is in terms of 24-hour periods, and is most useful when entered with either a plus sign (meaning "after") or a minus sign (meaning "before"). For example, consider the following scenario:

```
$ ls -l ?
-rw-------    1 root     root            0 Jan  7 18:00 a
-rw-------    1 root     root            0 Jan  6 18:00 b
-rw-------    1 root     root            0 Jan  5 18:00 c
-rw-------    1 root     root            0 Jan  4 18:00 d
$ date
Mon Jan  7 18:14:52 EST 2002
```

You could search for files that were created in the past 24 hours:

```
$ find . -name \? -mtime -1
./a
```

Or you could search for files that were created prior to the current 24-hour period:

```
$ find . -name \? -mtime +0
./b
./c
./d
```

The -daystart option

If you additionally specify the `-daystart` option, then the periods of time start at the beginning of today rather than 24 hours ago. For example, here is a set of files created yesterday and the day before:

```
$ find . -name \? -daystart -mtime +0 -mtime -3
./b
./c
$ ls -l b c
-rw-------    1 root     root            0 Jan  6 18:00 b
-rw-------    1 root     root            0 Jan  5 18:00 c
```

The -size option

The `-size` option allows you to find files based on their size. By default, the argument to `-size` is 512-byte blocks, but adding a suffix can make things easier. The available suffixes are b (512-byte blocks), c (bytes), k (kilobytes), and w (2-byte words). Additionally, you can prepend a plus sign ("larger than") or minus sign ("smaller than").

For example, to find regular files in `/usr/bin` that are smaller than 50 bytes:

```
$ find /usr/bin -type f -size -50c
```

```
/usr/bin/krdb
/usr/bin/run-nautilus
/usr/bin/sgmlwhich
/usr/bin/muttbug
```

Processing found files

You may be wondering what you can do with all these files that you find! Well, `find` has the ability to act on the files that it finds by using the `-exec` option. This option accepts a command line to execute as its argument, terminated with `;`; and replacing any occurrences of `{}` with the filename. This is best understood with an example:

```
$ find /usr/bin -type f -size -50c -exec ls -l '{}' ';'
-rwxr-xr-x    1 root     root           27 Oct 28 07:13 /usr/bin/krdb
-rwxr-xr-x    1 root     root           35 Nov 28 18:26 /usr/bin/run-nautilus
-rwxr-xr-x    1 root     root           25 Oct 21 17:51 /usr/bin/sgmlwhich
-rwxr-xr-x    1 root     root           26 Sep 26 08:00 /usr/bin/muttbug
```

As you can see, `find` is a very powerful command. It has grown through the years of UNIX and Linux development. There are many other useful options to `find`. You can learn about them in the `find` manual page.

locate

We have covered `which`, `whereis`, and `find`. You might have noticed that `find` can take a while to execute, due to the fact that it needs to read each directory that it's searching. It turns out that the `locate` command can speed things up by relying on an external database.

The `locate` command matches against any part of a pathname, not just the file itself. For example:

```
$ locate bin/ls
/var/ftp/bin/ls
/bin/ls
/sbin/lsmod
/sbin/lspci
/usr/bin/lsattr
/usr/bin/lspgpot
/usr/sbin/lsof
```

Using updatedb

Most Linux systems include a periodic process to update the database. If your system returned an error with the above command such as the following, then you will need to run `updatedb` to generate the search database:

```
$ locate bin/ls
locate: /var/spool/locate/locatedb: No such file or directory
$ su
Password:
```

```
# updatedb
```

The `updatedb` command may take a long time to run. If you have a noisy hard disk, you will hear a lot of racket as the entire filesystem is indexed. :)

slocate

In many Linux distributions, the `locate` command has been replaced by `slocate`. There is typically a symbolic link to "locate" so that you don't need to remember which you have. `slocate` stands for "secure locate". It stores permissions information in the database so that normal users can't pry into directories they wouldn't otherwise be able to read. The usage information for `slocate` is essentially the same as for `locate`, although the output might be different depending on the user running the command.

Section 4. Process control

Starting xeyes

To learn about process control, we first need to start a process:

```
$ xeyes -center red
```

You will notice that an `xeyes` window pops up, and the red eyeballs follow your mouse around the screen. You may also notice that you don't have a new prompt on your terminal.

Stopping a process

To get a prompt back, you could type Control-C (often written as Ctrl-C or ^C):

```
^C
$
```

You get a new `bash` prompt, but the `xeyes` window disappeared. In fact, the entire process has been killed. Instead of killing it with Control-C, we could have just stopped it with Control-Z:

```
$ xeyes -center red
^Z
[1]+  Stopped                 xeyes -center red
$
```

This time you get a new `bash` prompt, and the `xeyes` windows stays up. If you play with it a bit, however, you will notice that the eyeballs are frozen in place. If the `xeyes` window gets covered by another window and then uncovered again, you will see that it doesn't even redraw the eyes at all. The process isn't doing *anything*. It is, in fact, "Stopped".

---

fg and bg

To get the process "un-stopped" and running again, we can bring it to the foreground with the bash built-in `fg`:

```
$ fg
```

`xeyes`*`and`*`xeyes`

```
^Z
[1]+  Stopped                 xeyes -center red
$
```

Now continue it in the background with the bash built-in `bg`:

```
$ bg
[1]+ xeyes -center red &
```

```
$
```

Great! The `xeyes` process is now running in the background, and we have a new, working `bash` prompt.

Using "&"

If we wanted to start xeyes in the background from the beginning (instead of using Control-Z and `bg`), we could have just added an "&" (ampersand) to the end of xeyes command line:

```
$ xeyes -center blue &
[2] 16224
```

Multiple background processes

Now we have both a red and a blue `xeyes` running in the background. We can list these jobs with the bash built-in `jobs`:

```
$ jobs -l
[1]- 16217 Running                 xeyes -center red &
[2]+ 16224 Running                 xeyes -center blue &
```

The numbers in the left column are the job numbers bash assigned to these when they were started. Job 2 has a + (plus) to indicate that it's the "current job", which means that typing `fg` will bring it to the foreground. You could also foreground a specific job by specifying its number, in other words, `fg 1` would make the red `xeyes` the foreground task. The next column is the process id or `pid`, included in the listing, courtesy of the `-l` option to `jobs`. Finally, both jobs are currently "Running", and their command lines are listed to the right.

Introducing signals

To kill, stop, or continue processes, Linux uses a special form of communication called "signals". By sending a certain signal to a process, you can get it to terminate, stop, or do other things. This is what you're actually doing when you type Control-C, Control-Z, or use the `bg` or `fg` built-ins -- you're using `bash` to send a particular signal to the process. These signals can also be sent using the `kill` command and specifying the pid (process id) on the command line:

```
$ kill -s SIGSTOP 16224
$ jobs -l
[1]- 16217 Running                 xeyes -center red &
[2]+ 16224 Stopped (signal)        xeyes -center blue
```

As you can see, `kill` doesn't necessarily "kill" a process, although it can. Using the "-s" option, kill can send any signal to a process. Linux kills, stops, or continues processes when they are sent the SIGINT, SIGSTOP, or SIGCONT signals, respectively. There are also other signals that you can send to a process; some of these signals may be interpreted in an application-dependent way. You can learn what signals a particular process recognizes by

looking at its man page and searching for a `SIGNALS` section.

SIGTERM and SIGINT

If you *want* to kill a process, you have several options. By default, kill sends SIGTERM, which is not identical to SIGINT of Control-C fame, but usually has the same results:

```
$ kill 16217
$ jobs -l
[1]- 16217 Terminated              xeyes -center red
[2]+ 16224 Stopped (signal)        xeyes -center blue
```

The big kill

Processes can ignore both SIGTERM and SIGINT, either by choice or because they are stopped or somehow "stuck". In these cases it may be necessary to use the big hammer, the SIGKILL signal. A process cannot ignore SIGKILL:

```
$ kill 16224
$ jobs -l
[2]+ 16224 Stopped (signal)        xeyes -center blue
$ kill -s SIGKILL
$ jobs -l
[2]+ 16224 Interrupt               xeyes -center blue
```

nohup

The terminal where you start a job is called the job's controlling terminal. Some shells (not bash by default), will deliver a SIGHUP signal to backgrounded jobs when you logout, causing them to quit. To protect processes from this behavior, use the `nohup` when you start the process:

```
$ nohup make &
$ exit
```

Using ps to list processes

The `jobs` command we were using earlier only lists processes that were started from your bash session. To see all the processes on your system, use `ps` with the `a` and `x` options together:

```
$ ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      0:04 init [3]
    2 ?        SW     0:11 [keventd]
    3 ?        SWN    0:13 [ksoftirqd_CPU0]
    4 ?        SW     2:33 [kswapd]
    5 ?        SW     0:00 [bdflush]
```

We've listed only the first few because it's usually a very long list. This gives you a snapshot of what the whole machine is doing, but it is a lot of information to sift through. If you were to omit the `ax`, you would see only processes that are owned by you, and that have a controlling terminal. The command `ps x` would show you all your processes, even those without a controlling terminal. If you were to use `ps a`, you would get the list of everybody's processes that are attached to a terminal.

Seeing the forest and the trees

You can also list different information about each process. The `--forest` option makes it easy to see the process hierarchy, which will give you an indication of how the various processes on your system interrelate. When a process starts a new process, that new process is called a "child" process. In a `--forest` listing, parents appear on the left, and children appear as branches to the right:

```
$ ps x --forest
  PID TTY      STAT   TIME COMMAND
  927 pts/1    S      0:00 bash
 6690 pts/1    S      0:00  \_ bash
26909 pts/1    R      0:00      \_ ps x --forest
19930 pts/4    S      0:01 bash
25740 pts/4    S      0:04  \_ vi processes.txt
```

The "u" and "l" ps options

The "u" or "l" options can also be added to any combination of "a" and "x" in order to include more information about each process:

```
$ ps au
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
agriffis   403  0.0  0.0   2484    72 tty1     S     2001   0:00 -bash
chouser    404  0.0  0.0   2508    92 tty2     S     2001   0:00 -bash
root       408  0.0  0.0   1308   248 tty6     S     2001   0:00 /sbin/agetty 3
agriffis   434  0.0  0.0   1008     4 tty1     S     2001   0:00 /bin/sh /usr/X
chouser    927  0.0  0.0   2540    96 pts/1    S     2001   0:00 bash
```

```
$ ps al
  F   UID   PID  PPID PRI  NI    VSZ   RSS WCHAN  STAT TTY        TIME COMMAND
100  1001   403     1   9   0   2484    72 wait4  S    tty1       0:00 -bash
100  1000   404     1   9   0   2508    92 wait4  S    tty2       0:00 -bash
000     0   408     1   9   0   1308   248 read_c S    tty6       0:00 /sbin/ag
000  1001   434   403   9   0   1008     4 wait4  S    tty1       0:00 /bin/sh
000  1000   927   652   9   0   2540    96 wait4  S    pts/1      0:00 bash
```

Using "top"

If you find yourself running `ps` several times in a row, trying to watch things change, what you probably want is `top`. `top` displays a continuously updated process listing, along with some useful summary information:

```
$ top
```

```
 10:02pm  up 19 days,  6:24,  8 users,  load average: 0.04, 0.05, 0.00
75 processes: 74 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  1.3% user,  2.5% system,  0.0% nice, 96.0% idle
Mem:   256020K av,  226580K used,   29440K free,       0K shrd,    3804K buff
Swap:  136544K av,   80256K used,   56288K free                  101760K cached
  PID USER     PRI  NI  SIZE  RSS SHARE STAT  LIB %CPU %MEM   TIME COMMAND
  628 root      16   0  213M  31M  2304 S       0  1.9 12.5  91:43 X
26934 chouser   17   0  1272 1272  1076 R       0  1.1  0.4   0:00 top
  652 chouser   11   0 12016 8840  1604 S       0  0.5  3.4   3:52 gnome-termin
  641 chouser    9   0  2936 2808  1416 S       0  0.1  1.0   2:13 sawfish
```

nice

Each process has a priority setting that Linux uses to determine how fast it should run relative to the processes on the system. You can set the priority of a process by starting it with the `nice` command:

```
$ nice -n 10 oggenc /tmp/song.wav
```

Because the priority setting is called `nice`, it should be easy to remember that a higher value will be nice to other processes, allowing them to get priority access to the CPU. By default, processes are started with a setting of `0`, so the setting of `10` above means `oggenc` will readily give up the CPU to other processes. Generally, this means that `oggenc` will allow other processes to run at their normal speed, regardless of how CPU-hungry `oggenc` happens to be. You can see these niceness levels under the NI column in the `ps` and `top` listings above.

renice

The `nice` command can only change the priority of a process when you start it. If you want to change the niceness setting of a running process, use `renice`:

```
$ ps l 641
  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000  1000   641     1   9   0  5876 2808 do_sel S    ?          2:14 sawfish
$ renice 10 641
641: old priority 0, new priority 10
$ ps l 641
  F   UID   PID  PPID PRI  NI   VSZ  RSS WCHAN  STAT TTY        TIME COMMAND
000  1000   641     1   9  10  5876 2808 do_sel S    ?          2:14 sawfish
```

Section 5. Text processing

Redirection revisited

You can use the > operator to redirect the output of a command to a file, as follows:

```
$ echo "firstfile" > copyme
```

In addition to redirecting output to a file, we can also take advantage of a powerful shell feature called pipes. Using pipes, we can pass the output of one command to the input of another command. Consider the following example:

```
$ echo "hi there" | wc
      1       2       9
```

The | character is used to connect the output of the command on the left to the input of the command on the right. In the example above, the echo command prints out the string `hi there` followed by a linefeed. That output would normally appear on the terminal, but the pipe redirects it into the wc command, which displays the number of lines, words, and characters in its input.

A pipe example

Here is another simple example:

```
$ ls -s | sort -n
```

In this case, `ls -s` would normally print a listing of the current directory on the terminal, preceding each file with its size. But instead we've piped the output into `sort -n`, which sorts the output numerically. This is a really useful way to find large files in your home directory!

The following examples are more complex, but they demonstrate the power that can be harnessed using pipes. We're going to throw out some commands we haven't covered yet, but don't let that slow you down. Concentrate instead on understanding how pipes work so you can employ them in your daily Linux tasks.

The decompression pipeline

Normally to decompress and untar a file, you might do the following:

```
$ bzip2 -d linux-2.4.16.tar.bz2
$ tar xvf linux-2.4.16.tar
```

The downside of this method is that it requires the creation of an intermediate, uncompressed file on your disk. Since `tar` has the ability to read directly from its input (instead of specifying a file), we could produce the same end result using a pipeline:

```
$ bzip2 -dc linux-2.4.16.tar.bz2 | tar xvf -
```

Woohoo! Our compressed tarball has been extracted, and we didn't need an intermediate file.

A longer pipeline

Here's another pipeline example:

```
$ cat myfile.txt | sort | uniq | wc -l
```

We use cat to feed the contents of `myfile.txt` to the sort command. When the sort command receives the input, it sorts all input lines so that they are in alphabetical order, and then sends the output to `uniq`. `uniq` removes any duplicate lines, sending the scrubbed output to `wc -l`. We've seen the `wc` command earlier, but without command-line options. When given the `-l` option, it prints only the number of lines in its input, instead of also including words and characters. Try creating a couple of test files with your favorite text editor, and use this pipeline to see what results you get.

The text processing whirlwind begins

Now we embark on a whirlwind tour of the standard Linux text processing commands. Because we're covering a lot of material in this tutorial, we don't have the space to provide examples for every command. Instead, we encourage you to read each command's man page (by typing `man echo`, for example) and learn how each command and its options work by spending some time playing with each one. As a rule, these commands print the results of any text processing to the terminal rather than modifying any specified files.

After we take our whirlwind tour of the standard Linux text processing commands, we'll take a closer look at output and input redirection. So yes, there is light at the end of the tunnel. :)

`echo`

`echo` prints its arguments to the terminal. Use the `-e` option if you want to embed backslash escape sequences; for example `echo -e "foo\nfoo"` will print `foo`, then a newline, and then `foo` again. Use the `-n` option to tell echo to omit the trailing newline that is appended to the output by default.

cat, sort, and uniq

`cat`

`cat` prints the *contents* of the files specified as arguments to the terminal. This is handy as the first command of a pipeline, for example, `cat foo.txt | blah`.

`sort`

`sort` prints the contents of the file specified on the command line in alphabetical order. Of

course, sort also accepts piped input. Type `man sort` to familiarize yourself with its various options that control sorting behavior.

`uniq`

`uniq` takes an *already sorted* file or stream of data (via a pipeline) and removes duplicate lines.

wc, head, and tail

`wc`

`wc` prints out the number of lines, words, and bytes in the specified file or in the input stream (from a pipeline). Type `man wc` to learn how to fine-tune what counts are displayed.

`head`

`head` prints out the first ten lines of a file or stream. Use the `-n` option to specify how many lines should be displayed.

`tail`

Prints out the last ten lines of a file or stream. Use the `-n` option to specify how many lines should be displayed.

tac, expand, and unexpand

`tac`

`tac` is like `cat`, but prints all lines in reverse order, in other words, the last line is printed first.

`expand`

`expand` converts input tabs to spaces. Use the `-t` option to specify the tabstop.

`unexpand`

`unexpand` converts input spaces to tabs. Use the `-t` option to specify the tabstop.

cut, nl, and pr

`cut`

`cut` extracts character-delimited fields from each line of an input file or stream.

`nl`

`nl` adds a line number to every line of input. This is useful for printouts.

`pr`

`pr` breaks files into multiple pages of output; typically used for printing.

tr, sed, and awk

`tr`

`tr` is a character translation tool; it's used to map certain characters in the input stream to certain other characters in the output stream.

`sed`

`sed` is a powerful stream-oriented text editor. You can learn about `sed` in the following *developerWorks* articles:

* *Sed by example, Part 1: Get to know the powerful UNIX editor*
* *Sed by example, Part 2: Taking further advantage of the UNIX text editor*
* *Sed by example, Part 3: Data crunching, sed style*

`awk`

`awk` is a handy line-oriented text-processing language. To learn about `awk`, read the following IBM developerWorks articles:

* *Awk by example, Part 1: Intro to the great language with the strange name*
* *Awk by example, Part 2: Records, loops, and arrays*
* *Awk by example, Part 3: String functions and ... checkbooks?*

od, split, and fmt

`od`

`od` transforms the input stream into an octal or hex "dump" format.

`split`

`split` splits a larger file into many smaller-sized, more manageable chunks.

`fmt`

`fmt` reformats paragraphs so that wrapping is done at the margin. This ability is built into most text editors, but it's still a good one to know.

Paste, join, and tee

paste

`paste` takes two or more files as input, concatenates each sequential line from the input files, and outputs the resulting lines. It can be useful to create tables or columns of text.

`join`

`join` is similar to paste, but it uses a field (the first, by default) in each input line to match up what should be combined on a single line.

`tee`

`tee` prints its input both to a file and to the screen. This is useful when you want to create a log of something, but you also want to see it on the screen.

Whirlwind over! Redirection

Similar to using > on the bash command line, you can also use < to redirect a file *into* a command. For many commands, you can simply specify the filename on the command line, however some commands work only from standard input.

Bash and other shells support the concept of a "herefile". This allows you to specify the input to a command in the lines following the command invocation, terminating it with a sentinel value. Here's an example:

```
$ sort <<END
apple
cranberry
banana
END
apple
banana
cranberry
```

In our example, we typed the words `apple`, `cranberry`, and `banana`, followed by "END" to signify the end of the input. The `sort` program then returned our words in alphabetical order.

Using >>

You would expect >> to be somehow analogous to <<, but it isn't really. It simply means to append the output to a file, rather than overwrite as > would. For example:

```
$ echo Hi > myfile
$ echo there. > myfile
$ cat myfile
there.
```

Oops! We lost the "Hi" portion! We meant this:

```
$ echo Hi > myfile
$ echo there. >> myfile
$ cat myfile
Hi
```

```
there.
```

Much better!

Section 6. Resources and feedback

Resources and homework

Congratulations; you've reached the end of "Basic administration". We hope that this tutorial has helped you to firm up your foundational Linux knowledge. Please join us in our next tutorial, "Intermediate administration," where we'll build on the foundation laid here, covering topics like the Linux permissions and ownership model, user account management, filesystem creation and mounting, and more. And remember, by continuing in this tutorial series, you'll prepare yourself to attain your LPIC Level 1 Certification from the Linux Professional Institute. Speaking of LPIC certification, if this is something you're interested in, then we recommend that you study the following resources, which augment the material covered in this tutorial:

There are a number of good regular expression resources on the Web. Here are a couple that we've found:

* *Regular Expressions Reference*
* *Regular Expressions Explained*

Be sure to read up on the *Filesystem Hierarchy Standard*.

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:

* *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*

You can learn more about `sed` in the following *developerWorks* articles (Parts 1 and 2 will be good preparation for the LPIC Level 1 exam):

* *Sed by example, Part 1: Get to know the powerful UNIX editor*
* *Sed by example, Part 2: Taking further advantage of the UNIX text editor*
* *Sed by example, Part 3: Data crunching, sed style*

To learn more about `awk`, read the following *developerWorks* articles:

* *Awk by example, Part 1: Intro to the great language with the strange name*
* *Awk by example, Part 2: Records, loops, and arrays*
* *Awk by example, Part 3: String functions and ... checkbooks?*

We highly recommend the *Technical FAQ by Linux Users*, 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.

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.

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.

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$12345678901234567890123456789O1:11664:0:-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$12345678901234567890123456789O1:11664:0:-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$12345678901234567890123456789O1:11664:0:-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.

LPI certification 101 exam prep, Part 4

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 "Advanced Administration", the last of four tutorials designed to prepare you for the Linux Professional Institute's 101 exam. In this tutorial, we'll bolster your knowledge of important Linux administration skills by covering a variety of topics including: Linux filesystems, the Linux boot process, runlevels, filesystem quotas, and system logs.

By the end of this *series* of tutorials (eight in all), you'll 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.

Should I take this tutorial?

This Advanced administration tutorial (Part 4) is ideal for those who want to learn about or improve their Linux filesystem-related skills. This tutorial is particularly appropriate for someone who may be serving as the primary sysadmin for the first time, since we cover a lot of low-level issues that all system administrators should know. For many, 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 important Linux system administration skills and preparing for the next LPI certification level.

If you are new to Linux, we recommend that you complete *Part 1: Linux fundamentals*, *Part 2: Basic administration*, and *Part 3: Intermediate administration* 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. Linux filesystems

Block devices

In this section, we'll take a good look at Linux filesystems so that you're familiar with all the nitty-gritty details that an administrator needs to know. To begin, I'll introduce "block devices". The most famous block device is probably the one that represents the first IDE drive in a Linux system:

```
/dev/hda
```

If your system uses SCSI drives, then your first hard drive will be:

```
/dev/sda
```

The block devices above represent an *abstract* interface to the disk. User programs can use these block devices to interact with your disk without worrying about whether your drivers are IDE, SCSI or something else. The program can simply address the storage on the disk as a bunch of contiguous, randomly-accessible 512-byte blocks.

Whole disks and partitions

Under Linux, we create filesystems by using a special command called "mkfs", specifying a particular block device as a command-line argument.

However, although it is theoretically possible to use a "whole disk" block device (one that represents the *entire* disk) like /dev/hda or /dev/sda to house a single filesystem, this is almost never done in practice. Instead, full disk block devices are split up into smaller, more manageable block devices called "partitions". Partitions are created using a tool called **fdisk**, which is used to create and edit the partition table that's stored on each disk. The partition table defines exactly how to split up the full disk.

Introducing fdisk

We can take a look at a disk's partition table by running **fdisk**, specifying a block device that represents a full disk as an argument:

```
# fdisk /dev/hda
```

```
# fdisk /dev/sda
```

Note that you should *not* save or make any changes to a disk's partition table if any of its partitions contain filesystems that are in use or contain important data. Doing so will generally cause data on the disk to be lost.

Using fdisk

Once in fdisk, you'll be greeted with a prompt that looks like this:

```
Command (m for help):
```

Type **p** to display your disk's current partition configuration:

```
Command (m for help): p
Disk /dev/hda: 240 heads, 63 sectors, 2184 cylinders
Units = cylinders of 15120 * 512 bytes
Device Boot    Start       End    Blocks   Id  System
/dev/hda1             1        14    105808+  83  Linux
/dev/hda2            15        49    264600   82  Linux swap
/dev/hda3            50        70    158760   83  Linux
/dev/hda4            71      2184  15981840    5  Extended
/dev/hda5            71       209   1050808+  83  Linux
/dev/hda6           210       348   1050808+  83  Linux
/dev/hda7           349       626   2101648+  83  Linux
/dev/hda8           627       904   2101648+  83  Linux
/dev/hda9           905      2184   9676768+  83  Linux
Command (m for help):
```

This particular disk is configured to house seven Linux filesystems (listed as "Linux") as well as a swap partition (listed as "Linux swap"). Notice the name of the corresponding partition block devices on the left side, starting with /dev/hda1 and going up to /dev/hda9. In the early days of the PC, partitioning software only allowed a maximum of four partitions (called "primary" partitions). This was too limiting, so a workaround called an *extended partitioning* was created. An extended partition is very similar to a primary partition, and counts towards the primary partition limit of four. However, extended partitions can hold any number of so-called *logical* partitions inside them, providing an effective means of working around the four partition limit.

Using fdisk, continued

```
Command (m for help): p
Disk /dev/hda: 240 heads, 63 sectors, 2184 cylinders
Units = cylinders of 15120 * 512 bytes
Device Boot    Start       End    Blocks   Id  System
/dev/hda1             1        14    105808+  83  Linux
/dev/hda2            15        49    264600   82  Linux swap
/dev/hda3            50        70    158760   83  Linux
/dev/hda4            71      2184  15981840    5  Extended
/dev/hda5            71       209   1050808+  83  Linux
/dev/hda6           210       348   1050808+  83  Linux
/dev/hda7           349       626   2101648+  83  Linux
/dev/hda8           627       904   2101648+  83  Linux
/dev/hda9           905      2184   9676768+  83  Linux
Command (m for help):
```

In our example, hda1 through hda3 are primary partitions. hda4 is an extended partition that contains logical partitions hda5 through hda9. So, in this example, you would never actually *use* /dev/hda4 for storing any filesystems directly -- it simply acts as a container for partitions hda5 through hda9. Also, notice that each partition has an "Id", also called a "partition type". Whenever you create a new partition, you should ensure that the partition type is set correctly. '83' is the correct partition type for partitions that will be housing Linux filesystems,

and '82' is the correct partition type for Linux swap partitions. You set the partition type using the **t** option in fdisk. The Linux kernel uses the partition type setting to auto-detect filesystems and swap devices on the disk at boot-time.

Fdisk and beyond

There's more to fdisk than we have room to cover here, including the creation of new partitions (with the **n** command) and writing changes to disk (with the **w** command). Remember that you can type **m** for help. If you're new to **fdisk**, I recommend that you get the hang of the program by creating some partitions on spare disk where no data is at risk. Once you create your partitions and write them to disk, your new partition block devices are ready for use. In a bit, we'll use these new block devices to store new Linux filesystems.

For more information on partitioning, take a look at the following partitioning tips:

* *Partition planning tips*
* *Partitioning in action: consolidating data*
* *Partitioning in action: moving /home*

Creating filesystems

Before a new block device can be used for storing files, we need to create a new filesystem on it. We do this by using a **mkfs** command -- the particular **mkfs** we use depends on the type of filesystem that we'd like to create. In this example, we use **mke2fs** to create an ext2 filesystem on /dev/hdc6, an empty and unused partition block device:

```
# mke2fs /dev/hdc6
mke2fs 1.25 (20-Sep-2001)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
1537088 inodes, 3072423 blocks
153621 blocks (5.00%) reserved for the super user
First data block=0
94 block groups
32768 blocks per group, 32768 fragments per group
16352 inodes per group
Superblock backups stored on blocks:
32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
This filesystem will be automatically checked every 22 mounts or 180 days,
whichever comes first.  Use tune2fs -c or -i to override.
```

Above, **mke2fs** created an empty ext2 filesystem on /dev/hda6.

Mounting filesystems

Once the filesystem is created, we can mount it using the **mount** command:

```
# mount /dev/hdc6 /mnt
```

To mount a filesystem, specify the partition block device as a first argument and a "mountpoint" as a second argument. The new filesystem will be "grafted in" at the mountpoint. This also has the effect of "hiding" any files that were in the **/mnt** directory on the parent filesystem. Later, when the filesystem is unmounted, these files will reappear. After executing the mount command, any files added to /mnt will be stored on the new ext2 filesystem.

Creating and using swap

If we had just created a partition that is intended to be used as a swap device, we would initialize it by using the **mkswap** command, specifying the partition block device as an argument:

```
# mkswap /dev/hdc6
```

Unlike regular filesystems, swap partitions aren't mounted. Instead, they are enabled using the **swapon** command:

```
# swapon /dev/hdc6
```

Normally, your Linux system's startup scripts will take care of automatically enabling your swap partitions. Therefore, the **swapon** command is generally only needed when you need to immediately add some swap that you just created. To view the swap devices currently enabled, type **cat /proc/swaps**.

Viewing mounted filesystems

To see what filesystems are mounted, type **mount** by itself:

```
# mount
/dev/ide/host0/bus1/target0/lun0/part7 on / type xfs (rw,noatime,nodiratime)
proc on /proc type proc (rw)
none on /dev type devfs (rw)
tmpfs on /dev/shm type tmpfs (rw)
/dev/hdc6 on /mnt type ext2 (rw)
```

You can also view similar information by typing **cat /proc/mounts**. Since my Linux system uses devfs, the first line of my mount output lists a long path for the "root" partition block device. The "root" filesystem will get mounted automatically by the kernel at boot-time Systems that use the new devfs device-management filesystem for /dev have longer official names for the partition and disk block devices than Linux used to have in the past. For example, /dev/ide/host0/bus1/target0/lun0/part7 is the official name for /dev/hdc7, and /dev/hdc7 itself is just a symlink pointing to the official block device. You can determine if your system is using devfs by checking to see if the /dev/.devfsd file exists; if so, then devfs is active.

Mount options

It's possible to customize various attributes of the to-be-mounted filesystem by specifying *mount options*. For example, you can mount a filesystem as "read-only" by using the "ro" option:

```
# mount /dev/hdc6 /mnt -o ro
```

With /dev/hdc6 mounted read-only, no files can be modified in /mnt -- only read. If your filesystem is already mounted "read/write" and you want to switch it to read-only mode, you can use the "remount" option to avoid having to unmount and remount the filesystem again:

```
# mount /mnt -o remount,ro
```

Notice that we didn't need to specify the partition block device because the filesystem is already mounted and **mount** knows that /mnt is associated with /dev/hdc6. To make the filesystem writeable again, we can remount it as read-write:

```
# mount /mnt -o remount,rw
```

Note that these remount commands will not complete successfully if any process has opened any files or directories in /mnt. To familiarize yourself with all the mount options available under Linux, type **man mount**.

Introducing fstab

So far, we've seen how to mount filesystems manually. In general, if you have a filesystem that you need to mount on a regular basis, manual mounting tends to be a bit cumbersome. And for essential filesystems, such as a separate /var filesystem, manual mounting is not possible. These filesystems need to be mounted automatically at boot time, and we can tell the system to do just that by adding the appropriate entries to the /etc/fstab file. Even if you don't want the filesystem to be mounted automatically at boot, an /etc/fstab entry can make mounting easier, as we'll see in a bit.

A sample fstab

Let's take a look at a sample /etc/fstab file:

```
# <fs>                  <mountpoint>    <type>                <opts>
<dump/pass>
/dev/hda1         /boot         ext2         noauto,noatime                  1 2
/dev/hdc7         /         xfs         noatime,osyncisdsync,nodiratime 0 1
/dev/hdc5         none              swap         sw                          0 0
/dev/cdrom /mnt/cdrom iso9660         noauto,ro,user                 0 0
# /proc should always be enabled
proc         /proc             proc         defaults                          0 0
```

Above, each non-commented line in /etc/fstab specifies a partition block device, a mountpoint, a filesystem type, the filesystem options to use when mounting the filesystem,

and two numeric fields. The first numeric field is used to tell the **dump** backup command the filesystems that should be backed up. Of course, if you are not planning to use **dump** on your system, then you can safely ignore this field. The last field is used by the **fsck** filesystem integrity checking program, and tells it the order in which your filesystems should be checked at boot. We'll touch on **fsck** again in a few panels.

Look at the /dev/hda1 line; you'll see that /dev/hda1 is an ext2 filesystem that should be mounted at the /boot mountpoint. Now, look at /dev/hda1s mount options in the <opts> column. The **noauto** option tells the system to *not* mount /dev/hda1 automatically at boot time; without this option, /dev/hda1 would be automatically mounted to /boot at system boot time.

Also note the **noatime** option, which turns off the recording of **atime** (last access time) information on the disk. This information is generally not needed, and turning off atime updates has a positive effect on filesystem performance. You can also turn off directory atime updates by supplying the **nodiratime** mount option.

A sample fstab, continued

```
 # <fs>        <mountpoint>    <type>          <opts>
<dump/pass>
 /dev/hda1       /boot           ext2         noauto,noatime                  1 1
 /dev/hdc7       /               xfs         noatime,osyncisdsync,nodiratime 0 0
 /dev/hdc5       none            swap         sw                              0 0
 /dev/cdrom      /mnt/cdrom      iso9660 noauto,ro,user                      0 0
 # /proc should always be enabled
 proc            /proc           proc         defaults                          0 0
```

Now, take a look at the /proc line and notice the **defaults** option. Use **defaults** whenever you want a filesystem to be mounted with just the standard mount options. Since /etc/fstab has multiple fields, we can't simply leave the option field blank.

Also notice the /etc/fstab line for /dev/hdc5. This line defines /dev/hdc5 as a swap device. Since swap devices aren't mounted like filesystems, **none** is specified in the mountpoint field. Thanks to this /etc/fstab entry, our /dev/hdc5 swap device will be enabled automatically when the system starts up.

With an /etc/fstab entry for /dev/cdrom like the one above, mounting the CD-ROM drive becomes easier. Instead of typing:

```
 # mount -t iso9660 /dev/cdrom /mnt/cdrom -o ro
```

We can now type:

```
 # mount /dev/cdrom
```

In fact, using /etc/fstab allows us to take advantage of the **user** option. The **user** mount option tells the system to allow this particular filesystem to be mounted by any user. This comes in handy for removable media devices like CD-ROM drives. Without this fstab mount option, only the **root** user would be able to use the CD-ROM drive.

Unmounting filesystems

Generally, all mounted filesystems are unmounted automatically by the system when it is rebooted or shut down. When a filesystem is unmounted, any cached filesystem data in memory is flushed to the disk.

However, it's also possible to unmount filesystems manually. Before a filesystem can be unmounted, you first need to ensure that there are no processes running that have open files on the filesystem in question. Then, use the **umount** command, specifying *either* the device name or mount point as an argument:

```
# umount /mnt
```

or

```
# umount /dev/hdc6
```

Once unmounted, any files in /mnt that were "covered" by the previously-mounted filesystem will now reappear.

Introducing fsck

If your system crashes or locks up for some reason, the system won't have an opportunity to cleanly unmount your filesystems. When this happens, the filesystems are left in an inconsistent (unpredictable) state. When the system reboots, the **fsck** program will detect that the filesystems were not cleanly unmounted and will want to perform a consistency check of filesystems listed in /etc/fstab.

An important note -- for a filesystem to be checked by **fsck**, it must have a *non-zero* number in the "pass" field (the last field) in /etc/fstab. Typically, the root filesystem is set to a passno of **1**, specifying that it should be checked first. All other filesystems that should be checked at startup time should have a passno of **2** or higher.

Sometimes, you may find that after a reboot **fsck** is unable to fully repair a partially damaged filesystem. In these instances, all you need to do is to bring your system down to single-user mode and run **fsck** manually, supplying the partition block device as an argument. As **fsck** performs its filesystem repairs, it may ask you whether to fix particular filesystem defects. In general, you should say **y** (yes) to all these questions and allow **fsck** to do it's thing.

Problems with fsck

One of the problems with **fsck** scans is that they can take quite a while to complete, since the entirety of a filesystem's *metadata* (internal data structure) needs to be scanned in order to ensure that it's consistent. With extremely large filesystems, it is not unusual for an exhaustive **fsck** to take more than an hour.

In order to solve this problem, a new type of filesystem was designed, called a *journaling filesystem*. Journaling filesystems record an on-disk log of recent changes to the filesystem

metadata. In the event of a crash, the filesystem driver inspects the log. Because the log contains an accurate account of recent changes on disk, only these parts of the filesystem metadata need to be checked for errors. Thanks to this important design difference, checking a journalled filesystem for consistency typically takes just a matter of seconds, regardless of filesystem size. For this reason, journaling filesystems are gaining popularity in the Linux community. For more information on journaling filesystems, see the *Advanced filesystem implementor's guide, Part 1: Journalling and ReiserFS*.

Now, let's take a look at the various filesystems available for Linux.

The ext2 filesystem

The ext2 filesystem has been the standard Linux filesystem for many years. It has generally good performance for most applications, but it does not offer any journaling capability. This makes it unsuitable for very large filesystems, since fscks can take an extremely long time. In addition, ext2 has some built-in limitations due to the fact that every ext2 filesystem has a fixed number of inodes that it can hold. That being said, ext2 is generally considered to be an extremely robust and efficient non-journalled filesystem.

* In kernels: 2.0+
* journaling: no
* mkfs command: mke2fs
* mkfs example: mke2fs /dev/hdc7
* related commands: debugfs, tune2fs, chattr
* performance-related mount options: noatime, nodiratime

The ext3 filesystem

The ext3 filesystem uses the same on-disk format as ext2, but adds journaling capabilities. In fact, of all the Linux filesystems, ext3 has the most extensive journaling support, supporting not only metadata journaling but also ordered journaling (the default) and full metadata+data journaling. These "special" journaling modes help to ensure data integrity, not just short fscks like other journaling implementations. For this reason, ext3 is the best filesystem to use if data integrity is an absolute first priority. However, these data integrity features do impact performance somewhat. In addition, because ext3 uses the same on-disk format as ext2, it still suffers from the same scalability limitations as its non-journalled cousin. Ext3 is a good choice if you're looking for a good general-purpose journalled filesystem that is also very robust.

* In kernels: 2.4.16+
* journaling: metadata, ordered data writes, full metadata+data
* mkfs command: mke2fs -j
* mkfs example: mke2fs -j /dev/hdc7
* related commands: debugfs, tune2fs, chattr
* performance-related mount options: noatime, nodiratime
* other mount options:
    * data=writeback (disable journaling)
    * data=ordered (the default, metadata journaling and data is written out to disk with metadata)
    * data=journal (full data journaling for data *and* metadata integrity. Halves write

performance.)
* ext3 resources:
  * *Andrew Morton's ext3 page*
  * *Andrew Morton's excellent ext3 usage documentation (recommended)*
  * *Advanced filesystem implementor's guide, part 7: Introducing ext3*
  * *Advanced filesystem implementor's guide, part 8: Surprises in ext3*

The ReiserFS filesystem

ReiserFS is a relatively new filesystem that has been designed with the goal of providing very good small file performance, very good general performance and being very scalable. ReiserFS uses a metadata journal to avoid long fscks, but the journal implementation allows recently-modified data to become corrupted in the event of a system lockup. In general, ReiserFS offers very good performance, but may exhibit certain performance quirks under specific kinds of filesystem loads. In addition, ReiserFS's **fsck** tool is in its infancy so you may have difficulty recovering data from a corrupted filesystem. A number of these issues are due to the fact that ReiserFS is a relatively new, still evolving filesystem. ReiserFS is preferred by many for its speed and scalability.

* In kernels: 2.4.0+ (2.4.16+ recommended)
* journaling: metadata
* mkfs command: mkreiserfs
* mkfs example: mkreiserfs /dev/hdc7
* performance-related mount options: noatime, nodiratime, notail
* ReiserFS Resources:
  * *The home of ReiserFS*
  * *Advanced filesystem implementor's guide, Part 1: Journalling and ReiserFS*
  * *Advanced filesystem implementor's guide, Part 2: Using ReiserFS and Linux 2.4*

The XFS filesystem

The XFS filesystem is an enterprise-class journaling filesystem being ported to Linux by *SGI*. XFS isn't in the stock kernel yet, but more information on XFS can be found at *http://oss.sgi.com/projects/xfs*. For an introduction to XFS, read *Advanced filesystem implementor's guide, Part 9: Introducing XFS*.

The JFS filesystem

JFS is a high-performance journaling filesystem being ported to Linux by IBM. JFS is used by IBM enterprise servers and is designed for high-performance applications. JFS isn't in the stock kernel yet. You can learn more about JFS at *the JFS project Web site*.

VFAT

The VFAT filesystem isn't really a filesystem that you would choose for storing Linux files. Instead, it's a DOS-compatible filesystem driver that allows you to mount and exchange data with DOS and Windows FAT-based filesystems. The VFAT filesystem driver is present in the

standard Linux kernel.

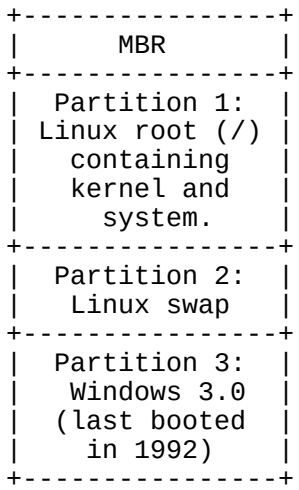
Section 3. Booting the system

About this section

This section introduces the Linux boot procedure. We'll cover the concept of a boot loader, how to set kernel options at boot, and how to examine the boot log for errors.

The MBR

The boot process is similar for all machines, regardless which distribution is installed. Consider the following example hard disk:



First, the computer's BIOS reads the first few sectors of your hard disk. These sectors contain a very small program, called the "Master Boot Record," or "MBR." The MBR has stored the location of the Linux kernel on the hard disk (partition 1 in the example above), so it loads the kernel into memory and starts it.

The kernel boot process

The next thing you see (although it probably flashes by quickly) is a line similar to the following:

```
Linux version 2.4.16 (root@time.flatmonk.org) (gcc version 2.95.3 20010315 (release)) #
```

This is the first line printed by the kernel when it starts running. The first part is the kernel version, followed by the identification of the user that built the kernel (usually root), the compiler that built it, and the timestamp when it was built.

Following that line is a whole slew of output from the kernel regarding the hardware in your system: the processor, PCI bus, disk controller, disks, serial ports, floppy drive, USB devices, network adapters, sound cards, and possibly others will each in turn report their status.

/sbin/init

When the kernel finishes loading, it starts a program called **init**. This program remains running until the system is shut down. It is always assigned process ID 1, as you can see:

```
$ ps --pid 1
  PID TTY          TIME CMD
    1 ?        00:00:04 init.system
```

The **init** program boots the rest of your distribution by running a series of scripts. These scripts typically live in /etc/rc.d/init.d or /etc/init.d, and they perform services such as setting the system's hostname, checking the filesystem for errors, mounting additional filesystems, enabling networking, starting print services, etc. When the scripts complete, init starts a program called **getty** which displays the login prompt, and you're good to go!

Digging in: LILO

Now that we've taken a quick tour through the booting process, let's look more closely at the first part: the MBR and loading the kernel. The maintenance of the MBR is the responsibility of the "boot loader". The two most popular boot loaders for x86-based Linux are "LILO" (LInux LOader) and "GRUB" (GRand Unified Bootloader).

Of the two, LILO is the older and more common boot loader. LILO's presence on your system is reported at boot, with the short "LILO boot:" prompt. Note that you may need to hold down the shift key during boot to get the prompt, since often a system is configured to whiz straight through without stopping.

There's not much fanfare at the LILO prompt, but if you press the <tab> key, you'll be presented with a list of potential kernels (or other operating systems) to boot. Often there's only one in the list. You can boot one of them by typing it and pressing <enter>. Alternatively you can simply press <enter> and the first item on the list will boot by default.

Using LILO

Occasionally you may want to pass an option to the kernel at boot time. Some of the more common options are **root=** to specify an alternative root filesystem, **init=** to specify an alternative init program (such as **init=/bin/sh** to rescue a misconfigured system), and **mem=** to specify the amount of memory in the system (for example **mem=512M** in the case that Linux only autodetects 128M). You could pass these to the kernel at the LILO boot prompt:

```
LILO boot: linux root=/dev/hdb2 init=/bin/sh mem=512M
```

If you need to regularly specify command-line options, you might consider adding them to /etc/lilo.conf. The format of that file is described in the *lilo.conf(5)* man page.

An important LILO gotcha

Before moving on to GRUB, there is an important gotcha to LILO. Whenever you make

changes to /etc/lilo.conf, or whenever you install a new kernel, you *must* run **lilo**. The **lilo** program rewrites the MBR to reflect the changes you made, including recording the absolute disk location of the kernel. The example here makes use of the **-v** flag for verboseness:

```
# lilo -v
LILO version 21.4-4, Copyright (C) 1992-1998 Werner Almesberger
'lba32' extensions Copyright (C) 1999,2000 John Coffman
Reading boot sector from /dev/hda
Merging with /boot/boot.b
Mapping message file /boot/message
Boot image: /boot/vmlinuz-2.2.16-22
Added linux *
/boot/boot.0300 exists - no backup copy made.
Writing boot sector.
```

Digging in: GRUB

The GRUB boot loader could be considered the next generation of boot loader, following LILO. Most visibly to users, it provides a menu interface instead of LILO's primitive prompt. For system administrators, the changes are more significant. GRUB supports more operating systems than LILO, provides some password-based security in the boot menu, and is easier to administrate.

GRUB is usually installed with the **grub-install** command. Once installed, GRUB's menu is administrated by editing the file /boot/grub/menu.lst. Both of these tasks are beyond the scope of this document; you should read the GRUB info pages before attempting to install or administrate GRUB.

Using GRUB

To give parameters to the kernel, you can press **e** at the boot menu. This provides you with the opportunity to edit (by again pressing **e**) either the name of the kernel to load or the parameters passed to it. When you're finished editing, press <enter> then **b** to boot with your changes.

A significant difference between LILO and GRUB that bears mentioning is that GRUB does not need to re-install its boot loader each time that the configuration changes, or a new kernel is installed. This is because GRUB understands the Linux filesystem, whereas LILO just stores the absolute disk location of the kernel to load. This single fact about GRUB alleviates the frustration system administrators feel when they forget to type **lilo** after installing a new kernel!

dmesg

The boot messages from the kernel and init scripts typically scroll by quickly. You might notice an error, but it's gone before you can properly read it. In that case, there are two places you can look after the system boots to see what went wrong (and hopefully get an idea how to fix it).

If the error occurred while the kernel was loading or probing hardware devices, you can

retrieve a copy of the kernel's log using the **dmesg** command:

```
# dmesg | head -1
Linux version 2.4.16 (root@time.flatmonk.org) (gcc version 2.95.3 20010315 (release)) #
```

Hey, we recognize that line! It's the first line the kernel prints when it loads. Indeed, if you pipe the output of dmesg into a pager, you can view all of the messages the kernel printed on boot, plus any messages the kernel has printed to the console in the meantime.

/var/log/messages

The second place to look for information is in the /var/log/messages file. This file is recorded by the syslog daemon, which accepts input from libraries, daemons, and the kernel. Each line in the messages file is timestamped. This file is a good place to look for errors that occurred during the init scripts stage of booting. For example, to see the last few messages from the nameserver:

```
# grep named /var/log/messages | tail -3
Jan 12 20:17:41 time /usr/sbin/named[350]: listening on IPv4 interface lo, 127.0.0.1#53
Jan 12 20:17:41 time /usr/sbin/named[350]: listening on IPv4 interface eth0, 10.0.0.1#5
Jan 12 20:17:41 time /usr/sbin/named[350]: running
```

Single-user mode

We know that it's possible to pass parameters to the kernel when it boots. One of the most often used parameters is **s** which causes the system to start in "single-user" mode. This mode usually mounts only the root filesystem, starts a minimal subset of the init scripts, and starts a shell rather than providing a login prompt. Additionally, networking is not configured, so there is no chance of external factors affecting your work.

Using single-user mode

So what "work" can be done with the system in such a state? To answer this question, we have to realize a vast difference between Linux and Windows. Windows is designed to normally be used by one person at a time, sitting at the console. It is effectively always in "single-user" mode. Linux, on the other hand, is used more often to serve network applications, or provide shell or X sessions to remote users on the network. These additional variables are not desirable when you want to perform maintenance operations such as restoring from backup, creating or modifying filesystems, upgrading the system from CD, etc. In these cases you should use single-user mode.

Changing runlevels

In fact, it's not actually necessary to reboot in order to reach single-user mode. The **init** program manages the current mode, or "runlevel", for the system. The standard runlevels for a Linux system are labeled and defined as follows:

* **0**: Halt the computer
* **1 or s**: Single-user mode
* **2**: Multi-user, no network
* **3**: Multi-user, text console
* **4**: Multi-user, graphical console
* **5**: Same as 4
* **6**: Reboot the computer

These runlevels vary between distributions, so be sure to consult your distro's documentation. To change to single-user mode, use the **telinit** command, which instructs init to change runlevels:

```
# telinit 1
```

You can see from the table above that you can also shutdown or reboot the system in this manner. **telinit 0** will halt the computer; **telinit 6** will reboot the computer. When you issue the **telinit** command to change runlevels, a subset of the init scripts will run to either shutdown or startup system services.

Shutting down gracefully

However, note that this is rather rude if there are users on the system at the time (who may be quite angry with you). The **shutdown** command provides a method for changing runlevels in a way that treats users reasonably. Similarly to the **kill** command's ability to send a variety of signals to a process, **shutdown** can be used to halt, reboot, or change to single-user mode. For example, to change to single-user mode in 5 minutes:

```
# shutdown 5
Broadcast message from root (pts/2) (Tue Jan 15 19:40:02 2002):
The system is going DOWN to maintenance mode in 5 minutes!
```

If you press control-c at this point, you can cancel the pending switch to single-user mode. The message above would appear on all terminals on the system, so users have a reasonable amount of time to save their work and log off. (Some might argue whether or not 5 minutes is "reasonable.")

Shutting down immediately

If you're the only person on the system, you can use "now" instead of an argument in minutes. For example, to reboot the system right now:

```
# shutdown -r now
```

No chance to hit control-c in this case; the system is already on its way down. Finally, the **-h** option halts the system:

```
# shutdown -h 1
Broadcast message from root (pts/2) (Tue Jan 15 19:50:58 2002):
The system is going DOWN for system halt in 1 minute!
```

The default runlevel

You've probably gathered at this point that the **init** program is quite important on a Linux system. You can configure init by editing the file /etc/initttab, which is described in the *inittab(5)* man page. We'll just touch on one key line in this file:

```
# grep ^id: /etc/inittab
id:3:initdefault:
```

On my system, runlevel 3 is the default runlevel. It can be useful to change this value if you prefer your system to boot immediately into a graphical login (usually runlevel 4 or 5). To do so, simply edit the file and change the value on that line. But be careful! If you change it to something invalid, you'll probably have to employ the **init=/bin/sh** trick we mentioned earlier.

Additional information

Additional information related to this section can be found at:

* IBM developerWorks' *Getting to know GRUB tutorial*
* *LILO Mini-HOWTO*
* *GRUB home*
* Kernel command-line options in /usr/src/linux/Documentation/kernel-parameters.txt
* *Sysvinit docs at Redhat*

Section 4. Runlevels

Single-user mode

Recall from the section regarding boot loaders that it's possible to pass parameters to the kernel when it boots. One of the most often used parameters is **s**, which causes the system to start in "single-user" mode. This mode usually mounts only the root filesystem, starts a minimal subset of the init scripts, and starts a shell rather than providing a login prompt. Additionally, networking is not configured, so there is no chance of external factors affecting your work.

Single-user mode, continued

So what "work" can be done with the system in such a state? To answer this question, we have to realize a vast difference between Linux and Windows. Windows is designed to normally be used by one person at a time, sitting at the console. It is effectively always in "single-user" mode. Linux, on the other hand, is used more often to serve network applications, or provide shell or X sessions to remote users on the network. These additional variables are not desirable when you want to perform maintenance operations such as restoring from backup, creating or modifying filesystems, upgrading the system from CD, etc. In these cases you should use single-user mode.

Runlevels

In fact, it's not actually necessary to reboot in order to reach single-user mode. The **init** program manages the current mode, or "runlevel", for the system. The standard runlevels for a Linux system are labeled and defined as follows:

* **0**: Halt the computer
* **1 or s**: Single-user mode
* **2**: Multi-user, no network
* **3**: Multi-user, text console
* **4**: Multi-user, graphical console
* **5**: Same as 4
* **6**: Reboot the computer

These runlevels vary between distributions, so be sure to consult your distro's documentation.

telinit

To change to single-user mode, use the **telinit** command, which instructs **init** to change runlevels:

```
# telinit 1
```

You can see from the table above that you can also shutdown or reboot the system in this manner. **telinit 0** will halt the computer; **telinit 6** will reboot the computer. When you issue

the **telinit** command to change runlevels, a subset of the init scripts will run to either shutdown or startup system services.

Runlevel etiquette

However, note that this is rather rude if there are users on the system at the time (who may be quite angry with you). The **shutdown** command provides a method for changing runlevels in a way that treats users reasonably. Similarly to the **kill** command's ability to send a variety of signals to a process, **shutdown** can be used to halt, reboot, or change to single-user mode. For example, to change to single-user mode in 5 minutes:

```
# shutdown 5
Broadcast message from root (pts/2) (Tue Jan 15 19:40:02 2002):
The system is going DOWN to maintenance mode in 5 minutes!
```

If you press control-c at this point, you can cancel the pending switch to single-user mode. The message above would appear on all terminals on the system, so users have a reasonable amount of time to save their work and log off. (Some might argue whether or not 5 minutes is "reasonable.")

"Now" and halt

If you're the only person on the system, you can use **now** instead of an argument in minutes. For example, to reboot the system right now:

```
# shutdown -r now
```

No chance to hit control-c in this case; the system is already on its way down. Finally, the **-h** option halts the system:

```
# shutdown -h 1
Broadcast message from root (pts/2) (Tue Jan 15 19:50:58 2002):
The system is going DOWN for system halt in 1 minute!
```

Configuring init

You've probably gathered at this point that the **init** program is quite important on a Linux system. You can configure init by editing the file /etc/initttab, which is described in the **inittab(5)** man page. We'll just touch on one key line in this file:

```
# grep ^id: /etc/inittab
id:3:initdefault:
```

On my system, runlevel 3 is the default runlevel. It can be useful to change this value if you prefer your system to boot immediately into a graphical login (usually runlevel 4 or 5). To do so, simply edit the file and change the value on that line. But be careful! If you change it to something invalid, you'll probably have to employ the **init=/bin/sh** trick we mentioned earlier.

Section 5. Filesystem quotas

Introducing quotas

Quotas are a feature of Linux that let you track disk usage by user or by group. They're useful for preventing any single user or group from using an unfair portion of a filesystem, or from filling it up altogether. Quotas can only be enabled and managed by the root user. In this section, I'll describe how to set up quotas on your Linux system and manage them effectively.

Kernel support

Quotas are a feature of the filesystem; therefore, they require kernel support. The first thing you'll need to do is verify that you have quota support in your kernel. You can do this using grep:

```
# cd /usr/src/linux
# grep -i quota .config
CONFIG_QUOTA=y
CONFIG_XFS_QUOTA=y
```

If this command returns something less conclusive (such as **CONFIG_QUOTA is not set**) then you should rebuild your kernel to include quota support. This is not a difficult process, but is outside of the scope of this section of the tutorial. If you're unfamiliar with the steps to build and install a new kernel, you might consider referencing *this tutorial*.

Filesystem support

Before diving into the administration of quotas, please note that quota support on Linux as of the 2.4.x kernel series is not complete. There are currently problems with quotas in the ext2 and ext3 filesystems, and ReiserFS does not appear to support quotas at all. This tutorial bases its examples on XFS, which *seems to properly support quotas*.

Configuring quotas

To begin configuring quotas on your system, you should edit /etc/fstab to mount the affected filesystems with quotas enabled. For our example, we use an XFS filesystem mounted with user and group quotas enabled:

```
# grep quota /etc/fstab
/usr/users   /mnt/hdc1    xfs    usrquota,grpquota,noauto   0 0
# mount /usr/users
```

Configuring quotas, continued

Note that the usrquota and grpquota options don't necessarily enable quotas on a filesystem. You can make sure quotas are enabled using the **quotaon** command:

```
# quotaon /usr/users
```

There is a corresponding **quotaoff** command should you desire to disable quotas in the future:

```
# quotaoff /usr/users
```

But for the moment, if you're trying some of the examples in this tutorial, be sure to have quotas enabled.

The quota command

The **quota** command displays a user's disk usage and limits for all of the filesystems currently mounted. The **-v** option includes in the list filesystems where quotas are enabled, but no storage is currently allocated to the user.

```
# quota -v
Disk quotas for user root (uid 0):
Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
 /dev/hdc1       0       0       0               3       0       0
```

The first column, blocks, shows how much disk space the root user is currently using on each filesystem listed. The following columns, quota and limit, refer to the limits currently in place for disk space. We will explain the difference between quota and limit, and the meaning of the grace column later on. The files column shows how many files the root user owns on the particular filesystem. The following quota and limit columns refer to the limits for files.

Viewing quota

Any user can use the **quota** command to view their own quota report as shown in the previous example. However only the root user can look at the quotas for other users and groups. For example, say we have a filesystem, /dev/hdc1 mounted on /usr/users, with two users: **jane** and **john**. First, let's look at jane's disk usage and limits.

```
# quota -v jane
Disk quotas for user jane (uid 1003):
Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
 /dev/hdc1    4100       0       0               6       0       0
```

In this example, we see that **jane**'s quotas are set to zero, which indicates no limit.

edquota

Now let's say we want to give the user **jane** a quota. We do this with the **edquota** command. Before we start editing quotas, let's see how much space we have available on /usr/users:

```
# df /usr/users
Filesystem           1k-blocks      Used Available Use% Mounted on
```

```
/dev/hdc1              610048      4276    605772   1% /usr/users
```

This isn't a particularly large filesystem, only 600M or so. It seems prudent to give **jane** a quota so that she can't use more than her fair share. When you run **edquota**, a temporary file is created for each user or group you specify on the command line.

edquota, continued

The **edquota** command puts you in an editor, which enables you to add and/or modify quotas via this temporary file.

```
# edquota jane
Disk quotas for user jane (uid 1003):
Filesystem         blocks       soft       hard     inodes     soft     hard
 /dev/hdc1           4100          0          0          6        0        0
```

Similar to the output from the quota command above, the blocks and inodes columns in this temporary file refer to the disk space and number of files jane is currently using. You cannot modify the number of blocks or inodes; any attempt to do so will be summarily discarded by the system. The soft and hard columns show jane's quota, which we can see is currently unlimited (zero indicates no quota).

Understanding edquota

The soft limit is the maximum amount of disk usage that jane has allocated to her on the filesystem (in other words, her quota). If jane uses more disk space than is allocated in her soft limit, she will be issued warnings about her quota violation via e-mail. The hard limit indicates the *absolute* limit on disk usage, which a user can't exceed. If jane tries to use more disk space than is specified in the hard limit, she will get a "Disk quota exceeded" error and will not be able to complete the operation.

Making changes

So here we change jane's soft and hard limits and save the file:

```
Disk quotas for user jane (uid 1003):
Filesystem         blocks       soft       hard     inodes     soft     hard
 /dev/hdc1           4100      10000      11500          6     2000     2500
```

Running the quota command, we can inspect our modifications:

```
# quota jane
Disk quotas for user jane (uid 1003):
Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
 /dev/hdc1    4100   10000   11500               6    2000    2500
```

Copying quotas

You'll remember that we also have another user john on this filesystem. If we want to give john the same quota as jane, we can use the **-p** option to edquota, which uses jane's quotas as a prototype for all following users on the command line. This is an easy way to set up quotas for groups of users.

```
# edquota -p jane john
# quota john
Disk quotas for user john (uid 1003):
Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
 /dev/hdc1       0   10000   11500               1    2000    2500
```

Group restrictions

We can also use edquota to restrict the allocation of disk space based on the group ownership of files. For example, to edit the quotas for the users group:

edquota -g users Disk quotas for group users (gid 100): Filesystem blocks soft hard inodes soft hard /dev/hdc1 4100 500000 510000 7 100000 125000
Then to view the modified quotas for the users group:

quota -g users Disk quotas for group users (gid 100): Filesystem blocks quota limit grace files quota limit grace /dev/hdc1 4100 500000 510000 7 100000 125000

The repquota command

Looking at each users' quotas using the quota command can be tedious if you have many users on a filesystem. The **repquota** command summarizes the quotas for a filesystem into a nice report. For example, to see the quotas for all users and groups on /usr/users:

```
# repquota -ug /usr/users
*** Report for user quotas on device /dev/hdc1
Block grace time: 7days; Inode grace time: 7days
                        Block limits                File limits
User            used    soft    hard  grace    used  soft  hard  grace
----------------------------------------------------------------------
root      --       0       0       0              3     0     0
john      --       0   10000   11500              1  2000  2500
jane      --    4100   10000   11500              6  2000  2500
*** Report for group quotas on device /dev/hdc1
Block grace time: 7days; Inode grace time: 7days
                        Block limits                File limits
Group           used    soft    hard  grace    used  soft  hard  grace
----------------------------------------------------------------------
root      --       0       0       0              3     0     0
users     --    4100  500000  510000              7 100000 125000
```

Repquota options

There are a couple other options to repquota that are worth mentioning. **repquota -a** will report on all currently-mounted read-write filesystems that have quotas enabled. **repquota -n** will not resolve uids and gids to names. This can speed up the output for large lists.

Monitoring quotas

If you are a system administrator, you will want to have a way to monitor quotas to ensure that they are not being exceeded. An easy way to do this is to use **warnquota**. The **warnquota** command sends e-mail to users who have exceeded their soft limit. Typically **warnquota** is run as a cron-job.

When a user exceeds their soft limit, the grace column in the output from the **quota** command will indicate the grace period -- how long before the soft limit is enforced for that filesystem.

```
Disk quotas for user jane (uid 1003):
    Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
    /dev/hdc1   10800*  10000   11500    7days      7     2000    2500
```

By default, the grace period for blocks and inodes is 7 days.

Modifying the grace period

You can modify the grace period for filesystems using equota:

```
# edquota -t
```

This puts you in an editor of a temporary file that looks like this:

```
Grace period before enforcing soft limits for users:
Time units may be: days, hours, minutes, or seconds
Filesystem             Block grace period     Inode grace period
/dev/hdc1                     7days                  7days
```

The text in the file is nicely explanatory. Be sure to leave your users enough time to receive their warning e-mail and find some files to delete!

Checking quotas on boot

You may also want to check quotas on boot. You can do this using a script to run the **quotacheck** command; there is an example script in the *Quota Mini HOWTO*. The **quotacheck** command also has the ability to repair damaged quota files; familiarize yourself with it by reading the quotacheck(8) man page.

Also remember what I mentioned previously regarding **quotaon** and **quotaoff.** You should incorporate **quotaon** into your boot script so that quotas are enabled. To enable quotas on all filesystems where quotas are supported, use the **-a** option:

```
# quotaon -a
```

Section 6. System logs

Introducing syslogd

The syslog daemon provides a mature client-server mechanism for logging messages from programs running on the system. Syslog receives a message from a daemon or program, categorizes the message by priority and type, then logs it according to administrator-configurable rules. The result is a robust and unified approach to managing logs.

Reading logs

Let's jump right in and look at the contents of a syslog-recorded log file. Afterward we can come back to syslog configuration. The FHS (see *Part 2* of this tutorial series) mandates that log files be placed in /var/log. Here we use the tail command to display the last 10 lines in the "messages" file:

```
# cd /var/log
# tail messages
Jan 12 20:17:39 bilbo init: Entering runlevel: 3
Jan 12 20:17:40 bilbo /usr/sbin/named[337]: starting BIND 9.1.3
Jan 12 20:17:40 bilbo /usr/sbin/named[337]: using 1 CPU
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: loading configuration from '/etc/bind/named
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: no IPv6 interfaces found
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: listening on IPv4 interface lo, 127.0.0.1#5
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: listening on IPv4 interface eth0, 10.0.0.1#
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: running
Jan 12 20:41:58 bilbo gnome-name-server[11288]: starting
Jan 12 20:41:58 bilbo gnome-name-server[11288]: name server starting
```

You hopefully remember from the text-processing whirlwind that the **tail** command displays the last lines in a file. In this case, we can see that the nameserver **named** was recently started on this system, which is named **bilbo**. If we were deploying IPv6, we might notice that **named** found no IPv6 interfaces, indicating a potential problem. Additionally, we can see that a user may have recently started *GNOME*, indicated by the presence of **gnome-name-server**.

Tailing log files

An experienced system administrator might use **tail -f** to follow the output to a log file as it occurs:

```
# tail -f /var/log/messages
```

For example, in the case of debugging our theoretical IPv6 problem, running the above command in one terminal while stopping and starting **named** would immediately display the messages from that daemon. This can be a useful technique when debugging. Some administrators even like to keep a constantly running **tail -f messages** in a terminal where they can keep an eye on system events.

Grepping logs

Another useful technique is to search a log file using the **grep** utility, described in *Part 2* of this tutorial series. In the above case, we might use **grep** to find where "named" behavior has changed:

```
# grep named /var/log/messages
```

Log overview

The following summarizes the log files typically found in /var/log and maintained by syslog:

* **messages**: Informational and error messages from general system programs and daemons
* **secure**: Authentication messages and errors, kept separate from "messages" for extra security
* **maillog**: Mail-related messages and errors
* **cron**: Cron-related messages and errors
* **spooler**: UUCP and news-related messages and errors

syslog.conf

As a matter of fact, now would be a good time to investigate the syslog configuration file, /etc/syslog.conf. (Note: If you don't have syslog.conf, keep reading for the sake of information, but you may be using an alternative syslog daemon.) Browsing that file, we see there are entries for each of the common log files mentioned above, plus possibly some other entries. The file has the format **facility.priority action**, where those fields are defined as follows:

syslog.conf, continued

facility

Specifies the subsystem that produced the message. The valid keywords for facility are auth, authpriv, cron, daemon, kern, lpr, mail, news, syslog, user, uucp, and local0 through local7.

priority

Specifies the minimum severity of the message, meaning that messages of this priority and higher will be matched by this rule. The valid keywords for priority are debug, info, notice, warning, err, crit, alert, and emerg.

action

The action field should be either a filename, tty (such as /dev/console), remote machine prefixed by @ , comma-separated list of users, or ***** to send the message to everybody logged on. The most common action is a simple filename.

Reloading and additional information

Hopefully this overview of the configuration file helps you to get a feel for the strength of the syslog system. You should read the **syslog.conf(5)** man page for more information prior to making changes. Additionally the **syslogd(8)** man page supplies lots more detailed information.

Note that you need to inform the syslog daemon of changes to the configuration file before they are put into effect. Sending it a SIGHUP is the right method, and you can use the **killall** command to do this easily:

```
# killall -HUP syslogd
```

A security note

You should beware that the log files written to by syslogd will be created by the program if they don't exist. Regardless of your current umask setting, the files will be created world-readable. If you're concerned about the security, you should chmod the files to be read-write by root only. Additionally, the **logrotate** program (described below) can be configured to create new log files with the appropriate permissions. The syslog daemon always preserves the current attributes of an existing log file, so you don't need to worry about it once the file is created.

logrotate

The log files in /var/log will grow over time, and potentially could fill the filesystem. It is advisable to employ a program such as **logrotate** to manage the automatic archiving of the logs. The **logrotate** program usually runs as a daily cron job, and can be configured to rotate, compress, remove, or mail the log files.

For example, a default configuration of logrotate might rotate the logs weekly, keeping 4 weeks worth of backlogs (by appending a sequence number to the filename), and compress the backlogs to save space. Additionally, the program can be configured to deliver a **SIGHUP** to **syslogd** so that the daemon will notice the now-empty log files and append to them appropriately.

For more information on **logrotate**, see the **logrotate(8)** man page, which contains a description of the program and the syntax of the configuration file.

Advanced topic -- klogd

Before moving away from syslog, I'd like to note a couple of advanced topics for ambitious readers. These tips may save you some grief when trying to understand syslog-related topics.

First, the syslog daemon is actually part of the sysklogd package, which contains a second daemon called **klogd**. It's **klogd**'s job to receive information and error messages from the

kernel, and pass them on to syslogd for categorization and logging. The messages received by klogd are exactly the same as those you can retrieve using the **dmesg** command. The difference is that dmesg prints the current contents of a ring buffer in the kernel, whereas klogd is passing the messages to syslogd so that they won't be lost when the ring wraps around.

Advanced topic -- alternate loggers

Second, there are alternatives to the standard sysklogd package. The alternatives attempt to be more efficient, easier to configure, and possibly more feature-rich than sysklogd. *Syslog-ng* and *Metalog* seem to be some of the more popular alternatives; you might investigate them if you find sysklogd doesn't provide the level of power you need.

Third, you can log messages in your scripts using the **logger** command. See the **logger(1)** man page for more information.

Section 7. Resources and feedback

We're done... almost!

Congratulations, you've reached the end of this tutorial! Well, almost. We were unable to include a few topics in our first four tutorials due to space limitations. Fortunately, we have a couple of good resources that will help you get up to speed on these topics in no time. Be sure to cover these particular tutorials if you are planning to get your LPIC level 1 certification.

On the important topic of system backups, we refer you to an IBM developerWorks Linux zone tutorial on the subject called *Backing up your Linux machines*. In this tutorial, you'll learn how to back up Linux systems using a **tar** variant called **star**. You'll also learn how to use the **mt** command to control tape functions.

The second topic that we weren't quite able to fit in was periodic scheduling. Fortunately, there's a good ***cron*** *and* ***at*** *tutorial over at thelinuxgurus.org*. **cron** and **at** are used to schedule jobs to be executed at a specific time, and are important knowledge for any system administrator.

In the next panel, you'll find a number of resources that you will find helpful in learning more about the subjects presented in this tutorial.

Resources

To find out more about quota support under Linux, be sure to check out the *Linux Quota mini-HOWTO* . Also be sure to consult the quota(1), edquota(8), repquota(8), quotacheck(8), and quotaon(8) man pages on your system.

Additional information to the system boot process and boot loaders can be found at:

* The IBM developerWorks Linux zone's *Getting to know GRUB tutorial*
* *LILO Mini-HOWTO*
* *GRUB home*
* Kernel command-line options in /usr/src/linux/Documentation/kernel-parameters.txt
* *Sysvinit docs at Redhat*

To learn more about Linux filesystems, read the multi-part advanced filesystem implementor's guide on the IBM developerWorks Linux zone, covering:

* *The benefits of journalling and ReiserFS* (Part 1)
* *Setting up a ReiserFS system* (Part 2)
* *Using the tmpfs virtual memory filesystem and bind mounts* (Part 3)
* *The benefits of devfs, the device management filesystem* (Part 4)
* *Beginning the conversion to devfs* (Part 5)
* *Completing the conversion to devfs using an init wrapper* (Part 6)
* *The benefits of the ext3 filesystem* (Part 7)
* *An in-depth look at ext3 and the latest kernel updates* (Part 8)
* *An introduction to XFS* (Part 9)

For more information on partitioning, take a look at the following partitioning tips on the IBM developerWorks Linux zone:

* *Partition planning tips*
* *Partitioning in action: consolidating data*
* *Partitioning in action: moving /home*

ReiserFS Resources:

* *The home of ReiserFS*
* *Advanced filesystem implementor's guide, Part 1: Journalling and ReiserFS* on developerWorks
* *Advanced filesystem implementor's guide, Part 2: Using ReiserFS and Linux 2.4* on developerWorks

ext3 resources:

* *Andrew Morton's ext3 page*
* *Andrew Morton's excellent ext3 usage documentation (recommended)*

XFS and JFS resources:

* *SGI XFS projects page*
* The IBM *JFS project Web site*

Don't forget *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.

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 excellent additional preparation for the LPIC Level 1 exam:

* *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.

LPI certification 102 exam prep, Part 1

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 "Compiling sources and managing packages," the first of four tutorials designed to prepare you for the Linux Professional Institute's 102 exam. In this tutorial, we'll show you how to compile programs from sources, how to manage shared libraries, and how to use the Red Hat and Debian package management systems.

By the end of this *series* of tutorials (eight in all), you'll 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.

Should I take this tutorial?

This tutorial on compiling sources and managing packages is ideal for those who want to learn about or improve their Linux package management skills. This tutorial is particularly appropriate for those who will be setting up applications on Linux servers or desktops. For many readers, much of this material will be new, but more experienced Linux users may find this tutorial to be a great way to "round out" their important Linux system administration skills.

If you are new to Linux, we recommend that you first complete the LPI certification 101 exam prep series of tutorials, which includes *Part 1: Linux fundamentals*, *Part 2: Basic administration*, *Part 3: Intermediate administration*, and *Part 4: Advanced administration* 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 he 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, "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, New Hampshire.

Section 2. Shared libraries

Introducing shared libraries

On Linux systems there are two fundamentally different types of Linux executable programs. The first are called *statically linked* executables. Static executables contain all the functions that they need to execute -- in other words, they're "complete." Because of this, static executables do not depend on any external library to run.

The second are *dynamically linked* executables. We'll get into those in the next panel.

Static vs. dynamic executables

We can use the `ldd` command to determine if a particular executable program is static:

```
# ldd /sbin/sln
        not a dynamic executable
```

"not a dynamic executable" is `ldd`'s way of saying that `sln` is statically linked. Now, let's take a look at `sln`'s size in comparison to its non-static cousin, `ln`:

```
# ls -l /bin/ln /sbin/sln
-rwxr-xr-x    1 root     root        23000 Jan 14 00:36 /bin/ln
-rwxr-xr-x    1 root     root       381072 Jan 14 00:31 /sbin/sln
```

As you can see, `sln` is over *ten* times the size of `ln`. `ln` is so much smaller than `sln` because it is a dynamic executable. Dynamic executables are *incomplete* programs that depend on external shared libraries to provide many of the functions that they need to run.

Dynamically linked dependencies

To view a list of all the shared libraries upon which `ln` depends, use the `ldd` command:

```
# ldd /bin/ln
        libc.so.6 => /lib/libc.so.6 (0x40021000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

As you can see, `ln` depends on the external shared libraries `libc.so.6` and `ld-linux.so.2`. As a rule, dynamically linked programs are much smaller than their statically-linked equivalents. However, statically-linked programs come in handy for certain low-level maintenance tasks. For example, `sln` is the perfect tool to modify various library symbolic links that exist in `/lib`. But in general, you'll find that nearly all executables on a Linux system are of the dynamically linked variety.

The dynamic loader

So, if dynamic executables don't contain everything they need to run, what part of Linux has

the job of loading them along with any necessary shared libraries so that they can execute correctly? The answer is something called the *dynamic loader*, which is actually the `ld-linux.so.2` library that you see listed as a shared library dependency in `ln`'s `ldd` listing. The dynamic loader takes care of loading the shared libraries that dynamically linked executables need in order to run. Now, let's take a quick look at how the dynamic loader finds the appropriate shared libraries on your system.

ld.so.conf

The dynamic loader finds shared libraries thanks to two files -- `/etc/ld.so.conf` and `/etc/ld.so.cache`. If you `cat` your `/etc/ld.so.conf` file, you'll probably see a listing that looks something like this:

```
$ cat /etc/ld.so.conf
/usr/X11R6/lib
/usr/lib/gcc-lib/i686-pc-linux-gnu/2.95.3
/usr/lib/mozilla
/usr/lib/qt-x11-2.3.1/lib
/usr/local/lib
```

The `ld.so.conf` file contains a listing of all directories (besides `/lib` and `/usr/lib`, which are automatically included) in which the dynamic loader will look for shared libraries.

ld.so.cache

But before the dynamic loader can "see" this information, it must be converted into an `ld.so.cache` file. This is done by running the `ldconfig` command:

```
# ldconfig
```

When ldconfig completes, you now have an up-to-date `/etc/ld.so.cache` file that reflects any changes you've made to `/etc/ld.so.conf`. From this point forward, the dynamic loader will look in any new directories that you specified in `/etc/ld.so.conf` when looking for shared libraries.

ldconfig tips

To view all the shared libraries that `ldconfig` can "see," type:

```
# ldconfig -p | less
```

There's one other handy trick you can use to configure your shared library paths. Sometimes, you'll want to tell the dynamic loader to try to use shared libraries in a specific directory before trying any of your `/etc/ld.so.conf` paths. This can be handy in situations where you are running an older application that doesn't work with the currently-installed versions of your libraries.

LD_LIBRARY_PATH

To instruct the dynamic loader to check a certain directory first, set the `LD_LIBRARY_PATH` variable to the directories that you would like searched. Separate multiple paths using commas; for example:

```
# export LD_LIBRARY_PATH="/usr/lib/old:/opt/lib"
```

After `LD_LIBRARY_PATH` has been exported, any executables started from the current shell will use libraries in `/usr/lib/old` or `/opt/lib` if possible, falling back to the directories specified in `/etc/ld.so.conf` if some shared library dependencies are still unsatisfied.

We've completed our coverage of Linux shared libraries. To learn more about shared libraries, type `man ldconfig` and `man ld.so`.

Section 3. Compiling applications from sources

Introduction

Let's say you find a particular application that you'd like to install on your system. Maybe you need to run a very recent version of this program, but this most recent version isn't yet available in a packaging format such as rpm. Perhaps this particular application is only available in source form, or you need to enable certain features of the program that are not enabled in the rpm by default.

Whatever the reason, whether of necessity or simply just because you *want* to compile the program from its sources, this section will show you how.

Downloading

Your first step will be to locate and download the sources that you want to compile. They'll probably be in a single archive with a trailing .tar.gz, tar.Z, tar.bz2, or .tgz extension. Go ahead and download the archive with your favorite browser or ftp program. If the program happens to have a Web page, this would be a good time to visit it to familiarize yourself with any installation documentation that may be available.

The program you're installing could depend on the existence of any number of other programs that may or may not be currently installed on your system. If you know for sure that your program depends on other programs or libraries that are not currently installed, you'll need to get these packages installed first (either from a binary package like rpm or by compiling them from their sources also.) Then, you'll be in a great position to get your original source file successfully installed.

Unpacking

Unpacking the source archive is relatively easy. If the name of your archive ends with `.tar.gz`, `.tar.Z`, or `.tgz`, you should be able to unpack the archive by typing:

```
$ tar xzvf archivename.tar.gz
```

(`x` is for extract, `z` is for gzip decompression, `v` is for verbose (print the files that are extracted), and `f` means that the filename will appear next on the command line.)

Nearly all "source tarballs" will create one main directory that contains all the program's sources. This way, when you unpack the archive, your current working directory isn't cluttered with lots of files -- instead, all files are neatly organized in a single directory and don't get in the way.

Listing archives

Every now and then, you may come across an archive that, when decompressed, creates tons of files in your current working directory. While most tarballs aren't created this way, it's been known to happen. If you want to verify that your particular tarball was put together

correctly and creates a main directory to hold the sources, you can view its contents by typing:

```
$ tar tzvf archivename.tar.gz | more
```

(`t` is for a *text* listing of the archive. No extraction occurs.)

If there is no common directory listed on the left-hand side of the archive listing, you'll want to create a new directory, move the tarball inside it, enter the directory, and only then extract the tarball. Otherwise, you'll be in for a mess!

Unpacking bzip2-compressed archives

It's possible that your archive may be in `.tar.bz2` format. Archives with this extension have been compressed with bzip2. Bzip2 generally compresses significantly better than gzip. Its only disadvantage is that compression and decompression are slower, and bzip2 consumes more memory than gzip while running. For modern computers, this isn't much of an issue, which is why you can expect bzip2 to become more and more popular as time goes on.

Because bzip2 has been gaining popularity, many Linux distributions come with versions of tar that have been patched so that passing a `y` or `i` option will inform tar that the archive is in bzip2 format and needs to be automatically decompressed with the bzip2 program. To see if you have a patched version of tar, try typing:

```
$ tar tyvf archive.tar.bz2 | more
```

or

```
$ tar tivf archive.tar.bz2 | more
```

If neither of these commands work (and tar complains of an invalid argument), there is still hope -- read on.

bzip2 pipelines

So, your version of tar doesn't recognize those handy bzip2 shortcuts -- what can be done? Fortunately, there's an easy way to extract the contents of bzip2 tarballs that will work on nearly all UNIX systems, even if the system in question happens to have a non-GNU version of tar. To view the contents of a bzip2 file, we can create a pipeline:

```
$ cat archive.tar.bz2 | bzip2 -d | tar tvf - | most
```

This next pipeline will actually extract the contents of archive.tar.bz2:

```
$ cat archive.tar.bz2 | bzip2 -d | tar xvf -
```

bzip2 pipelines (continued)

In the previous two examples, we created a standard UNIX pipeline to view and extract files from our archive file. Since tar was called with the `f -` option, it read tar data from stdin,

rather than trying to read data from a file on disk.

If you used the pipeline method to try to extract the contents of your archive and your system complained that bzip2 couldn't be found, it's possible that bzip2 isn't installed on your system. You can download the sources to bzip2 from *http://sourceware.cygnus.com/bzip2*. After installing the bzip2 sources (by following this tutorial), you'll then be able to unpack and install the application you wanted to install in the first place :)

Inspecting sources

Once you've unpacked your sources, you'll want to enter the unpacked directory and check things out. It's always a good idea to locate any installation-related documentation. Typically, this information can be found in a `README` or `INSTALL` file located in the main source directory. Additionally, look for `README.platform` and `INSTALL.platform` files, where platform is the name of your particular operating system.

Configuration

Many modern sources contain a *configure* script in the main source directory. This script (typically generated by the developers using the GNU autoconf program) is specially designed to set up the sources so that they compile perfectly on your system. When run, the configure script probes your system, determining its capabilities, and creates *Makefiles*, which contain instructions for building and installing the sources on your system.

The configure script is almost always called "configure." If you find a configure script in the main source directory, odds are good that it was put there for your use. If you can't find a configure script, then your sources probably come with a standard Makefile that has been designed to work across a variety of systems -- this means that you can skip the following configuration steps, and resume this tutorial where we start talking about "make."

Using configure

Before running the configure script, it's a good idea to get familiar with it. By typing `./configure --help`, you can view all the various configuration options that are available for your program. Many of the options you see, especially the ones listed at the top of the `--help` printout, are standard options that will be found in nearly every configure script. The options listed near the end are often related to the particular package you're trying to compile. Take a look at them and note any you'd like to enable or disable.

The --prefix option

Most GNU autoconf-based configure scripts have a `--prefix` option that allows you to control where your program is installed. By default, most sources install into the `/usr/local` prefix. This means that binaries end up in `/usr/local/bin`, man pages in `/usr/local/man`, etc. This is normally what you want; `/usr/local` is commonly used to store programs that you compile yourself.

Using --prefix

If you'd like the sources to install somewhere else, say in `./usr`, you'll want to pass the `--prefix=/usr` option to configure. Likewise, you could also tell configure to install to your `/opt` tree, by using the `--prefix=/opt` option.

What about FHS?

Sometimes, a particular program may default to installing some of its files to non-standard locations on disk. In particular, a source archive may have a number of installation paths that do not follow the Linux Filesystem Hierarchy Standard (FHS). Fortunately, the configure script doesn't just permit the changing of the install prefix, but also allows us to change the install location for various system components such as man pages.

This capability comes in very handy, since most source archives aren't yet FHS-compliant. Nearly always, you'll need to add a `--mandir=/usr/share/man` and a `--infodir=/usr/share/info` to the configure command line in order to make your source package FHS-compliant.

Time to configure

Once you've taken a look at the various configure options and determined which ones you'd like to use, it's time to run configure. Please note that you may not *need* to include any command-line options when you run configure -- in the majority of situations, the defaults will work (but may not be *exactly* what you want).

Time to configure (continued)

To run configure, type:

```
$ ./configure <options>
```

This could look like:

```
$ ./configure
```

or

```
$ ./configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --with-th
```

The options you need will depend on the particular package you're configuring. When you run configure, it will spend a minute or two detecting what particular features or tools are available on your system, printing out the results of its various configuration checks as it goes.

config.cache

Once the configuration process completes, the configure script stores all its configuration data in a file called `config.cache`. This file lives in the same directory as the configure script itself. If you ever need to run `./configure` again after you've updated your system configuration, make sure you `rm config.cache` first; otherwise, configure will simply use the old settings without rechecking your system.

configure and Makefiles

After the configure script completes, it's time to compile the sources into a running program. A program called *make* is used to perform this step. If your software package contained a configure script, then when you ran it, configure created Makefiles that were specially customized for your system. These files tell the make program how to build the sources and install the resultant binaries, man pages, and support files.

Makefile intro

Makefiles are typically named `makefile` or `Makefile`. There will normally be one makefile in each directory that contains source files, in addition to one that sits in the main source directory. The autoconf-generated Makefiles contain instructions (officially called *rules*) that specify how to build certain *targets*, like the program you want to install. make figures out the order in which all the rules should run.

Invoking make

Invoking make is easy; just type "make" in the current directory. The make program will then find and interpret a file called `makefile` or `Makefile` in the current directory. If you type "make" all by itself, it will build the *default target*. Developers normally set up their makefiles so that the default target compiles all the sources:

```
$ make
```

Some makefiles won't have a default target, and you'll need to specify one in order to get the compilation started:

```
$ make all
```

After typing one of these commands, your computer will spend several minutes compiling your program into object code. Presuming it completes with no errors, you'll be ready to install the compiled program onto your system.

Installation

After the program is compiled, there's one more important step: installation. Although the program is compiled, it's not yet ready for use. All its components need to be copied from the source directory to the correct "live" locations on your filesystem. For example, all binaries need to be copied to `/usr/local/bin`, and all man pages need to be installed into

`/usr/local/man`, etc.

Before you can install the software, you'll need to become root. This is typically done by either logging in as root on a separate terminal or typing "su," at which point you'll be prompted for root's password. After typing it in, you'll have root privileges until you exit from your current shell session by typing "exit" or hitting control-D. If you're already root, you're ready to go!

make install

Installing sources is easy. In the main source directory, simply type:

```
# make install
```

Typing "make install" will tell make to satisfy the "install" target; this target is traditionally used to copy all the freshly created source files to the correct locations on disk so that your program can be used. If you didn't specify a `--prefix` option, it's very likely that quite a few files and directories will be copied to your `/usr/local` tree. Depending on the size of the program, the install target may take anywhere from several seconds to a few minutes to complete.

In addition to simply copying files, make install will also make sure the installed files have the correct ownership and permissions. After `make install` completes successfully, the program is installed and ready (or *almost* ready) for use!

Once it's installed

Now that your program is installed, what's next? Running it, of course! If you're not familiar with how to use the program you just installed, you'll want to read the program's man page by typing:

```
$ man programname
```

It's possible that a program may require additional configuration steps. For example, if you installed a Web server, you'll need to configure it to start automatically when your system boots. You may also need to customize a configuration file in `/etc` before your application will run.

Ta da!

Now that you've fully installed a particular software package from its sources, you can now run it! To start the program, type:

```
$ programname
```

Congratulations!

Possible problems

It's very possible that configure or make, or possibly even make install, aborted with some kind of error code. The next several panels will help you correct common problems.

Missing libraries

Every now and then, you may experience a problem where configure bombs out because you don't have a certain library installed. In order for you to continue the build process, you'll need to temporarily put your current program configuration on hold and track down the sources or binary package for the library that your program needs. Once the correct library is installed, configure or make should be happy and complete successfully.

Other problems

Sometimes, you'll run into some kind of error that you simply don't know how to fix. As your experience with UNIX/Linux grows, you'll be able to diagnose more and more seemingly cryptic error conditions that you encounter during the configure and make process.

Sometimes, errors occur because an installed library is too old (or possibly even too new!). Other times, the problem you're having is actually the fault of the developers, who may not have anticipated their program running on a system such as yours -- or maybe they just made a typo :)

Other problems (continued)

For problems such as these, use your best judgment to determine where to go for help. If this is your first attempt at compiling a program from source, this may be a good time to select another, easier program to compile. Once you get the simpler program compiled, you may have the necessary experience to fix your originally encountered problem. As you continue to learn more about how UNIX works, you'll get closer to the point where you can actually "tweak" Makefiles and sources to get even seemingly flaky code to compile cleanly!

Section 4. Package management concepts

Package management advantages

Beyond building applications from sources, there's another method for installing software on your Linux system. All Linux distributions employ some form of package management for installing, upgrading, and uninstalling software packages. Package management offers clear advantages over installing directly from source:

- Ease of installation and uninstallation
- Ease of upgrading already-installed packages
- Protection of configuration files
- Simple tracking of installed files

Package management disadvantages

Before jumping into instructions for using the most popular package management tools, I'll acknowledge that there are some Linux users who dislike package management. They might propose some of the following downsides:

- Binaries built for a specific system perform better
- Resolving package dependencies is a headache
- Package database corruption can render a system unmaintainable
- Packages are hard to create

There is some truth to these statements, but the general consensus among Linux users is that the advantages outweigh the disadvantages. Additionally, each stumbling block listed above has a corresponding rebuttal: Multiple packages can be built to optimize for different systems; package managers can be augmented to resolve dependencies automatically; databases can be rebuilt based on other files; and the initial effort expended in creating a package is mitigated by the ease of upgrading or removing that package later.

Section 5. rpm, the (R)ed Hat (P)ackage (M)anager

Getting started with rpm

The introduction of Red Hat's *rpm* in 1995 was a huge step forward for Linux distributions. Not only did it make possible the management of packages on *Red Hat Linux*, but due to its GPL license, rpm has become the defacto standard for open source packaging.

The rpm program has a command-line interface by default, although there are GUIs and Web-based tools to provide a friendlier interface. In this section we'll introduce the most common command-line operations, using the *Xsnow* program for the examples. If you would like to follow along, you can download rpm below, which should work on most rpm-based distributions.

- *xsnow-1.41-1.i386.rpm*

Note: If you find the various uses of the term "rpm" confusing in this section, keep in mind that "rpm" usually refers to the program, whereas "an rpm" or "the rpm" usually refers to an rpm package.

Installing an rpm

To get started, let's install our `Xsnow` rpm using `rpm -i`:

```
# rpm -i xsnow-1.41-1.i386.rpm
```

If this command produced no output, then it worked! You should be able to run Xsnow to enjoy a blizzard on your X desktop. Personally, we prefer to have some visual feedback when we install an rpm, so we like to include the `-h` (hash marks to indicate progress) and `-v` (verbose) options:

```
# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow                       ##################################################
```

Re-installing an rpm

If you were following along directly, you might have seen the following message from rpm in the previous example:

```
# rpm -ivh xsnow-1.41-1.i386.rpm
package xsnow-1.41-1 is already installed
```

There may be occasions when you wish to re-install an rpm, for instance if you were to accidentally delete the binary `/usr/X11R6/bin/xsnow`. In that case, you should first remove the rpm with `rpm -e`, then re-install it. Note that the information message from rpm in the following example does not hinder the removal of the package from the system:

```
# rpm -e xsnow
removal of /usr/X11R6/bin/xsnow failed: No such file or directory

# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow                       ##################################################
```

Forcefully installing an rpm

Sometimes removing an rpm isn't practical, particularly if there are other programs on the system that depend on it. For example, you might have installed an "x-amusements" rpm which lists Xsnow as a dependency, so using `rpm -e` to remove Xsnow is disallowed:

```
# rpm -e xsnow
error: removing these packages would break dependencies:
        /usr/X11R6/bin/xsnow is needed by x-amusements-1.0-1
```

In that case, you could re-install Xsnow using the `--force` option:

```
# rpm -ivh --force xsnow-1.41-1.i386.rpm
xsnow                       ##################################################
```

Installing or removing with --nodeps

An alternative to using `--force` in the previous panel would be to remove the rpm using the `--nodeps` option. This disables rpm's internal dependency checking, and is *not recommended* in most circumstances. Nonetheless, it is occasionally useful:

```
# rpm -e --nodeps xsnow

# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow                       ##################################################
```

You can also use `--nodeps` when installing an rpm. To re-iterate what was said above, using `--nodeps` is not recommended, however it is sometimes necessary:

```
# rpm -ivh --nodeps xsnow-1.41-1.i386.rpm
xsnow                       ##################################################
```

Upgrading packages

Eventually there will probably be an rpm of Xsnow version 1.42, which is available on the Xsnow author's *Website*. When that occurs, you'll want to upgrade your existing Xsnow installation. If you were to use `rpm -ivh --force`, it would appear to work, but rpm's internal database would list *both* versions as being installed. Instead, you should use `rpm -U` to upgrade your installation:

```
# rpm -Uvh xsnow-1.42-1.i386.rpm
xsnow                       ##################################################
```

Here's a little trick: we rarely use `rpm -i` at all, because `rpm -U` will simply install an rpm if it doesn't exist yet on the system. This is especially useful if you specify multiple packages on the command-line, where some are currently installed and some are not:

```
# rpm -Uvh xsnow-1.42-1.i386.rpm xfishtank-2.1tp-1.i386.rpm
xsnow                       ##################################################
xfishtank                   ##################################################
```

Querying with rpm -q

You might have noticed in the examples that installing an rpm requires the full filename, but removing an rpm requires only the name. This is because rpm maintains an internal database of the currently installed packages, and you can reference installed packages by name. For example, let's ask rpm what version of Xsnow is installed:

```
# rpm -q xsnow
xsnow-1.41-1
```

In fact, rpm knows even more about the installed package than just the name and version. We can ask for a lot more information about the Xsnow rpm using `rpm -qi`:

```
# rpm -qi xsnow
Name        : xsnow                         Relocations: (not relocateable)
Version     : 1.41                               Vendor: Dan E. Anderson http://www.dan.d
Release     : 1                             Build Date: Thu 10 May 2001 01:12:26 AM EDT
Install date: Sat 02 Feb 2002 01:00:43 PM EST      Build Host: danx.drydog.com
Group       : Amusements/Graphics           Source RPM: xsnow-1.41-1.src.rpm
Size        : 91877                            License: Copyright 1984, 1988, 1990, 1993
Packager    : Dan E. Anderson http://dan.drydog.com/
URL         : http://www.euronet.nl/~rja/Xsnow/
Summary     : An X Window System based dose of Christmas cheer.
Description :
The Xsnow toy provides a continual gentle snowfall, trees, and Santa
Claus flying his sleigh around the screen on the root window.
Xsnow is only for the X Window System, though; consoles just get coal.
```

Listing files with rpm -ql

The database maintained by rpm contains quite a lot of information. We've already seen that it keeps track of what versions of packages are installed, and their associated information. It can also list the files owned by a given installed package using `rpm -ql`:

```
# rpm -ql xsnow
/etc/X11/applnk/Games/xsnow.desktop
/usr/X11R6/bin/xsnow
/usr/X11R6/man/man1/xsnow.1x.gz
```

Combined with the `-c` option or the `-d` option, you can restrict the output to configuration or documentation files, respectively. This type of query is more useful for larger rpms with long file lists, but we can still demonstrate using the Xsnow rpm:

```
# rpm -qlc xsnow
/etc/X11/applnk/Games/xsnow.desktop

# rpm -qld xsnow
/usr/X11R6/man/man1/xsnow.1x.gz
```

Querying packages with rpm -qp

If you had the information available with `rpm -qi` *before* installing the package, you might have been able to better decide whether or not to install it. Actually, using `rpm -qp` allows you to query an rpm file instead of querying the database. All of the queries we've seen so far can be applied to rpm files as well as installed packages. Here are all the examples again, this time employing the `-p` option:

```
# rpm -qp xsnow-1.41-1.i386.rpm
xsnow-1.41-1

# rpm -qpi xsnow-1.41-1.i386.rpm
[same output as rpm -qi in the previous panel]

# rpm -qpl xsnow-1.41-1.i386.rpm
/etc/X11/applnk/Games/xsnow.desktop
/usr/X11R6/bin/xsnow
/usr/X11R6/man/man1/xsnow.1x.gz

# rpm -qplc xsnow-1.41-1.i386.rpm
/etc/X11/applnk/Games/xsnow.desktop

# rpm -qpld xsnow-1.41-1.i386.rpm
/usr/X11R6/man/man1/xsnow.1x.gz
```

Querying all installed packages

You can query all the packages installed on your system by including the `-a` option. If you pipe the output through `sort` and into a pager, then it's a nice way to get a glimpse of what's installed on your system. For example:

```
# rpm -qa | sort | less
[output omitted]
```

Here's how many rpms we have installed on one of our systems:

```
# rpm -qa | wc -l
    287
```

And here's how many files are in all those rpms:

```
# rpm -qal | wc -l
  45706
```

Here's a quick tip: Using `rpm -qa` can ease the administration of multiple systems. If you

redirect the sorted output to a file on one machine, then do the same on the other machine, you can use the `diff` program to see the differences.

Finding the owner for a file

Sometimes it's useful to find out what rpm owns a given file. In theory, you could figure out what rpm owns `/usr/X11R6/bin/xsnow` (pretend you don't remember) using a shell construction like the following:

```
# rpm -qa | while read p; do rpm -ql $p | grep -q '^/usr/X11R6/bin/xsnow$' && echo $p; d
xsnow-1.41-1
```

Since this takes a long time to type, and even longer to run (1m50s on one of our Pentiums), the rpm developers thoughtfully included the capability in rpm. You can query for the owner of a given file using `rpm -qf`:

```
# rpm -qf /usr/X11R6/bin/xsnow
xsnow-1.41-1
```

Even on the Pentium, that only takes 0.3s to run. And even fast typists will enjoy the simplicity of `rpm -qf` compared to the complex shell construction :)

Showing dependencies

Unless you employ options such as `--nodeps`, rpm normally won't allow you to install or remove packages that break dependencies. For example, you can't install Xsnow without first having the X libraries on your system. Once you have Xsnow installed, you can't remove the X libraries without removing Xsnow first (and probably half of your installed packages).

This is a strength of rpm, even if it's frustrating sometimes. It means that when you install an rpm, it should just *work*. You shouldn't need to do much extra work, since rpm has already verified that the dependencies exist on the system.

Sometimes when you're working on resolving dependencies, it can be useful to query a package with the `-R` option to learn about everything it expects to be on the system. For example, the Xsnow package depends on the C library, the math library, the X libraries, and specific versions of rpm:

```
# rpm -qpR xsnow-1.41-1.i386.rpm
rpmlib(PayloadFilesHavePrefix) <= 4.0-1
ld-linux.so.2
libX11.so.6
libXext.so.6
libXpm.so.4
libc.so.6
libm.so.6
libc.so.6(GLIBC_2.0)
libc.so.6(GLIBC_2.1.3)
rpmlib(CompressedFileNames) <= 3.0.4-1
```

You can also query the installed database for the same information by omitting the `-p`:

```
# rpm -qR xsnow
```

Verifying the integrity of a package

When you download an rpm from the Web or an ftp site, for the sake of security you may want to verify its integrity before installing. All rpms are "signed" with an MD5 sum. Additionally, some authors employ a PGP or GPG signature to further secure their packages. To check the signature of a package, you can use the `--checksig` option:

```
# rpm --checksig xsnow-1.41-1.i386.rpm
xsnow-1.41-1.i386.rpm: md5 GPG NOT OK
```

Wait a minute! According to that output, the GPG signature is *NOT OK*. Let's add some verbosity to see what's wrong:

```
# rpm --checksig -v xsnow-1.41-1.i386.rpm
xsnow-1.41-1.i386.rpm:
MD5 sum OK: 8ebe63b1dbe86ccd9eaf736a7aa56fd8
gpg: Signature made Thu 10 May 2001 01:16:27 AM EDT using DSA key ID B1F6E46C
gpg: Can't check signature: public key not found
```

So, the problem is that we couldn't retrieve the author's public key. After we retrieve the public key from the *package author's Website* (shown in the output from `rpm -qi`), the signature checks out:

```
# gpg --import dan.asc
gpg: key B1F6E46C: public key imported
gpg: /root/.gnupg/trustdb.gpg: trustdb created
gpg: Total number processed: 1
gpg:               imported: 1
```

```
# rpm --checksig xsnow-1.41-1.i386.rpm
xsnow-1.41-1.i386.rpm: md5 gpg OK
```

Verifying an installed package

Similarly to checking the integrity of an rpm, you can also check the integrity of your installed files using `rpm -V`. This step makes sure that the files haven't been modified since they were installed from the rpm:

```
# rpm -V xsnow
```

Normally this command displays no output to indicate a clean bill of health. Let's spice things up and try again:

```
# rm /usr/X11R6/man/man1/xsnow.1x.gz
```

```
# cp /bin/sh /usr/X11R6/bin/xsnow

# rpm -V xsnow
S.5....T   /usr/X11R6/bin/xsnow
missing    /usr/X11R6/man/man1/xsnow.1x.gz
```

This output shows us that the Xsnow binary fails MD5 sum, file size, and mtime tests. And the man page is missing altogether! Let's repair this broken installation:

```
# rpm -e xsnow
removal of /usr/X11R6/man/man1/xsnow.1x.gz failed: No such file or directory

# rpm -ivh xsnow-1.41-1.i386.rpm
xsnow                       ##################################################
```

Configuring rpm

Rpm rarely needs configuring. It simply works out of the box. In older versions of rpm, you could change things in `/etc/rpmrc` to affect run-time operation. In recent versions, that file has been moved to `/usr/lib/rpm/rpmrc`, and is not meant to be edited by system administrators. Mostly it just lists flags and compatibility information for various platforms (e.g. i386 is compatible with all other x86 architectures).

If you wish to configure rpm, you can do so by editing `/etc/rpm/macros`. Since this is rarely necessary, we'll let you read about it in the rpm bundled documentation. You can find the right documentation file with the following command:

```
# rpm -qld rpm | grep macros
```

Additional rpm resources

That's all we've got in this tutorial regarding the Red Hat Package Manager. You should have enough information to administer a system, and there are a lot more resources available. Be sure to check out some of these links:

- *rpm Home Page*
- *Maximum RPM - an entire book*
- *The RPM HOWTO at the Linux Documentation Project*
- *Red Hat's chapter on package management with rpm*
- *developerWorks article on creating rpms*
- *rpmfind.net -- a huge collection of rpms*

Section 6. Debian package management

Introducing apt-get

The Debian package management system is made up of several different tools. The command-line tool apt-get is the easiest way to install new packages. For example, to install the program Xsnow, do this as the root user:

```
# apt-get install xsnow
Reading Package Lists... Done
Building Dependency Tree... Done
The following NEW packages will be installed:
  xsnow
0 packages upgraded, 1 newly installed, 0 to remove and 10 not upgraded.
Need to get 17.8kB of archives. After unpacking 42.0kB will be used.
Get:1 http://ftp-mirror.internap.com stable/non-free xsnow 1.40-6 [17.8kB]
Fetched 17.8kB in 0s (18.4kB/s)
Selecting previously deselected package xsnow.
(Reading database ... 5702 files and directories currently installed.)
Unpacking xsnow (from .../archives/xsnow_1.40-6_i386.deb) ...
Setting up xsnow (1.40-6) ...
```

Skimming through this output, you can see that Xsnow was to be installed, then it was fetched it from the Web, unpacked, and finally set up.

Simulated install

If apt-get notices that the package you are trying to install depends on other packages, it will automatically fetch and install those as well. In the last example, only Xsnow was installed, because all of it's dependencies were already satisfied.

Sometimes, however, the list of packages apt-get needs to fetch can be quite large, and it is often useful to see what is going to installed before you let it start. The `-s` option does exactly this. For example, on one of our systems if we try to install the graphical e-mail program `balsa`:

```
# apt-get -s install balsa
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
  esound esound-common gdk-imlib1 gnome-bin gnome-libs-data imlib-base libart2
  libaudiofile0 libesd0 libglib1.2 libgnome32 libgnomesupport0 libgnomeui32
  libgnorba27 libgnorbagtk0 libgtk1.2 libjpeg62 liborbit0 libpng2 libproplist0
  libtiff3g libungif3g zlib1g
The following NEW packages will be installed:
  balsa esound esound-common gdk-imlib1 gnome-bin gnome-libs-data imlib-base
  libart2 libaudiofile0 libesd0 libglib1.2 libgnome32 libgnomesupport0
  libgnomeui32 libgnorba27 libgnorbagtk0 libgtk1.2 libjpeg62 liborbit0 libpng2
  libproplist0 libtiff3g libungif3g zlib1g
0 packages upgraded, 24 newly installed, 0 to remove and 10 not upgraded.
```

It then goes on to list the order in which the packages will be installed and configured (or set up).

Package resource list: apt-setup

Since apt-get is automatically fetching packages for you, it must know something about where to find packages that haven't yet been installed. This knowledge is kept in `/etc/apt/sources.list`. Although you can edit this file by hand (see the `sources.list` man page), you may find it easier to use an interactive tool:

```
# apt-setup
```

This tool walks you through the process of finding places to get Debian packages, such as CDROMs, Web sites, and ftp sites. When you're done, it writes out changes to your `/etc/apt/sources.list` file, so that apt-get can find packages when you ask for them.

From apt-get to dselect

The apt-get tool has many command-line options that you can read about in the apt-get man page. The defaults usually work fine, but if you find yourself using the same option frequently, you may want to add a setting to your `/etc/apt/apt.conf` file. This syntax for this configuration file is described in the `apt.conf` man page.

apt-get also has many other commands besides the `install` command we've used so far. One of these is `apt-get dselect-upgrade`, which obeys the Status set for each package on your Debian system.

Starting dselect

The Status for each package is stored in the file `/var/lib/dpkg/status`, but it is best updated using another interactive tool:

```
# dselect
Debian GNU/Linux `dselect' package handling frontend.

 * 0. [A]ccess      Choose the access method to use.
   1. [U]pdate      Update list of available packages, if possible.
   2. [S]elect      Request which packages you want on your system.
   3. [I]nstall     Install and upgrade wanted packages.
   4. [C]onfig      Configure any packages that are unconfigured.
   5. [R]emove      Remove unwanted software.
   6. [Q]uit        Quit dselect.

Move around with ^P and ^N, cursor keys, initial letters, or digits;
Press <enter> to confirm selection.   ^L redraws screen.

Version 1.6.15 (i386).  Copyright (C) 1994-1996 Ian Jackson.   This is
free software; see the GNU General Public License version 2 or later for
copying conditions.  There is NO warranty.  See dselect --license for details.
```

Using dselect Select mode

You can view and change each package's Status by choosing the Select option. It will then display a screenful of help. When you're done reading this, press space. Now you will see a list of packages that looks something like this:

```
EIOM Pri Section  Package      Inst.ver    Avail.ver   Description
     All packages
       Newly available packages
         New Important packages
           New Important packages in section admin
  n* Imp admin    at           <none>      3.1.8-10    Delayed job execution and
  n* Imp admin    cron         <none>      3.0pl1-57.3 management of regular bac
  n* Imp admin    logrotate    <none>      3.2-11      Log rotation utility
           New Important packages in section doc
  n* Imp doc      info         <none>      4.0-4       Standalone GNU Info docum
  n* Imp doc      manpages     <none>      1.29-2      Man pages about using a L
           New Important packages in section editors
  n* Imp editors  ed           <none>      0.2-18.1    The classic unix line edi
  n* Imp editors  nvi          <none>      1.79-16a.1  4.4BSD re-implementation
           New Important packages in section interpreters
  n* Imp interpre perl-5.005   <none>      5.005.03-7. Larry Wall's Practical Ex
           New Important packages in section libs
  n* Imp libs     libident     <none>      0.22-2      simple RFC1413 client lib
  n* Imp libs     libopenldap- <none>      1.2.12-1    OpenLDAP runtime files fo
  n* Imp libs     libopenldap1 <none>      1.2.12-1    OpenLDAP libraries.
  n* Imp libs     libpcre2     <none>      2.08-1      Philip Hazel's Perl Compa
```

The package Status

The Status for each package can be seen under the somewhat cryptic heading `EIOM`. The column we care about is under the `M` character, where each package is marked with one of the following:

To change the Mark, just press the key for the code you want (equal, dash, or underline), but if you want to change the Mark to `*` (asterisk), you have to press + (plus).

When you are done, use an upper-case `Q` to save your changes and exit the Select screen. If you need help at any time in `dselect`, type ? (question mark). Type a space to get back out of a help screen.

Install and Configure (dpkg-reconfigure)

Debian doesn't install or remove packages based on their Status settings until you run something like `apt-get dselect-upgrade`. This command actually does several steps for you at once -- Install, Remove, and Configure. The Install and Remove steps shouldn't need to stop to ask you any questions. The Configure step, however, may ask any number of questions in order to set up the package just the way you want it.

There are other ways to run these steps. For example, you can choose each step individually from the main `dselect` menu.

Some packages use a system called `debconf` for their Configure step. Those that do can ask their setup questions in a variety of ways, such as in a text terminal, through a graphical

interface, or through a Web page. To configure one of these packages, use the `dpkg-reconfigure` command. You can even use it to make sure *all* `debconf` packages have been completely configured:

```
# dpkg-reconfigure --all
debconf: package "3c5x9utils" is not installed or does not use debconf
debconf: package "3dchess" is not installed or does not use debconf
debconf: package "9menu" is not installed or does not use debconf
debconf: package "9wm" is not installed or does not use debconf
debconf: package "a2ps" is not installed or does not use debconf
debconf: package "a2ps-perl-ja" is not installed or does not use debconf
debconf: package "aalib-bin" is not installed or does not use debconf
```

This will produce a very long list of packages that do not use debconf, but it will also find some that do and present easy-to-use forms for you to answer the questions that each package asks.

Getting the status of an installed package

The Debian package management tools we've reviewed so far are best for handling multi-step operations with long lists of packages. But they don't cover some of the nuts-and-bolts operations of package management. For this kind of work, you want to use `dpkg`.

For example, to get the complete status and description of a package, use the `-s` option:

```
# dpkg -s xsnow
Package: xsnow
Status: install ok installed
Priority: optional
Section: non-free/x11
Installed-Size: 41
Maintainer: Martin Schulze <joey@debian.org>
Version: 1.40-6
Depends: libc6, xlib6g (>= 3.3-5)
Description: Brings Christmas to your desktop
 Xsnow is the X-windows application that will let it snow on the
 root window, in between and on windows.  Santa and his reindeer
 will complete your festive-season feeling.
```

The link between a file and its .deb

Since a `.deb` package contains files, you would think there would be a way to list the files within the package. Well, you would be right; just use the `-L` option:

```
# dpkg -L xsnow
/.
/usr
/usr/doc
/usr/doc/xsnow
/usr/doc/xsnow/copyright
/usr/doc/xsnow/readme.gz
/usr/doc/xsnow/changelog.Debian.gz
```

```
/usr/X11R6
/usr/X11R6/bin
/usr/X11R6/bin/xsnow
/usr/X11R6/man
/usr/X11R6/man/man6
/usr/X11R6/man/man6/xsnow.6.gz
```

To go the other way around, and find which package contains a specific file, use the `-S` option:

```
# dpkg -S /usr/doc/xsnow/copyright
xsnow: /usr/doc/xsnow/copyright
```

The name of the package is listed just to the left of the colon.

Finding packages to install

Usually, apt-get will already know about any Debian package you might need. If it doesn't, you may be able to find the package among these *lists of Debian packages*, or elsewhere on the Web.

If you do find and download a `.deb` file, you can install it using the `-i` option:

```
# dpkg -d /tmp/dl/xsnow_1.40-6_i386.deb
```

If you can't find the package you're looking for as a `.deb` file, but you find a `.rpm` or some other type of package, you may be able to use *alien*. The alien program can convert packages from various formats into `.debs`.

Additional Debian package management resources

There is a lot more to the Debian package management system than we covered here. There is also a lot more to Debian than its package management system. The following sites will help round out your knowledge in these areas:

- *Debian home page*
- *Debian installation guide*
- *Lists of Debian packages*
- *Alien home page*
- *Guide to creating your own Debian packages*

Section 7. Resources and feedback

Resources

Don't forget *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.

In the *Bash by example* article series, we show you how to use `bash` programming constructs to write your own `bash` scripts. This series (particularly parts one and two) will be excellent additional preparation for the LPIC Level 1 exam:

- *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 for 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 (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 *Linux glossary for Linux users*, also from Mark.

If you're not too familiar with the `vi` editor, we strongly recommend that you check out IBM's *Vi -- 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.

LPI certification 102 exam prep, Part 2

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 "Configuring and compiling the Linux kernel," the second of four tutorials designed to prepare you for the Linux Professional Institute's 102 exam. In this tutorial, we'll show you how to compile the Linux kernel from sources. Along the way, we'll cover various important kernel configuration options and provide more in-depth information about PCI and USB support in the kernel. By the end of this series of tutorials (eight in all; this is part six), you'll 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.

Should I take this tutorial?

This particular tutorial is ideal for those who want to learn about or improve their Linux kernel compilation and configuration skills. This tutorial is particularly appropriate for those who will be setting up Linux servers or desktops. For many, 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 important Linux kernel skills.

If you are new to Linux, we recommend that you first complete the previous tutorials in the LPI certification 101 and 102 exam prep series before continuing:

- *101 series, Part 1: Linux fundamentals*
- *101 series, Part 2: Basic administration*
- *101 series, Part 3: Intermediate administration*
- *101 series, Part 4: Advanced administration*
- *102 series, Part 1: Compiling sources and managing packages*

About the author

For technical questions about the content of this tutorial, contact the author:

- Daniel Robbins, at *drobbins@gentoo.org*

Residing in Albuquerque, New Mexico, Daniel Robbins 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 to 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.

Section 2. Introducing the kernel

And the kernel is... Linux!

Typically, the word "Linux" is used to refer to a complete Linux distribution and all the cooperating programs that make the distribution work. However, you may be surprised to find out that, technically, Linux is a kernel, and a kernel only. While the other parts of what we commonly call "Linux" (such as a shell and compiler) are essential parts of a complete operating environment, they are technically separate from Linux (the kernel). Still, people will continue to use the word "Linux" to mean "Linux-based distribution." Nevertheless, everyone can at least agree that the Linux kernel is the **heart** of every "Linux OS."

Interfacing with hardware

The primary role of the Linux kernel is to interface directly with the hardware in your system. The kernel provides a *layer of abstraction* between the raw hardware and application programs. This way, the programs themselves do not need to know the details of your specific motherboard chipset or disk controller -- they can instead operate at the higher level of reading and writing files to disk, for example.

CPU abstraction

The Linux kernel also provides a level of abstraction on top of the processor(s) in your system -- allowing for multiple programs to appear to run simultaneously. The kernel takes care of giving each process a fair and timely share of the processors' computing resources.

If you're running Linux right now, then the kernel that you are using now is either UP (uniprocessor) or SMP (symmetric multiprocessor) aware. If you happen to have an SMP motherboard, but you're using a UP kernel, Linux won't "see" your extra processors! To fix this, you'll want to compile a special SMP kernel for your hardware. Currently, SMP kernels will also work on uniprocessor systems, but at a slight performance hit.

Abstracting IO

The kernel also handles the much-needed task of abstracting all forms of file IO. Imagine what would happen if every program had to interface with your specific disk hardware directly -- if you changed disk controllers, all your programs would stop working! Fortunately, the Linux kernel follows the UNIX model of providing a simple data storage and access abstraction that all programs can use. That way, your favorite database doesn't need to be concerned whether it is storing data on an IDE disk, on a SCSI RAID array, or on a network-mounted file system.

Networking central

One of Linux's main claims to fame is its robust networking, especially TCP/IP support. And, if you guessed that the TCP/IP stack is in the Linux kernel, you're right! The kernel provides a standards-compliant, high-level interface for programs that want to send data over the

network. Behind the scenes, the Linux kernel interfaces directly with your particular ethernet card or `pppd` daemon and handles the low-level Internet communication details. Note that the next tutorial in this series, Part 7, will deal with TCP/IP and networking.

Networking goodies

One of the greatest things about Linux is the wealth of useful optional features that are available in the kernel, especially related to networking. For example, you can configure a kernel that will allow your entire home network to access the Internet by way of your Linux modem -- this is called IP Masquerading, or IP NAT.

Additionally, the Linux kernel can be configured to export or mount network-based NFS file systems, allowing for other UNIX machines on your LAN to easily share data with your Linux system. There are a lot of goodies in the kernel, as you'll learn once you begin exploring the Linux kernel's many configuration options.

Booting review

Now would be a good time for a quick refresher of the Linux boot process. When you turn on your Linux-based system, the kernel image (stored in a single binary file) is loaded from disk to memory by a boot loader, such as LILO or GRUB. At this point, the kernel takes control of your system. One of the first things it does is detect and initialize all the hardware that it finds and has been configured to support. Once the hardware has been initialized properly, the kernel is ready to start normal user-space programs (also known as "processes").

The first process run by the kernel is /sbin/init. It, in turn, starts additional processes as specified in /etc/inittab. Within seconds, your Linux system is up and running, ready for you to use. Although you never interact with the kernel directly, the Linux kernel is always running "above" all normal processes, providing the necessary virtualization and abstractions that your various programs and libraries require to function.

Introducing... modules!

All recent Linux kernels support kernel modules. Kernel modules are really neat things -- they're pieces of the kernel that reside in relatively small binary files on disk. As soon as the kernel needs the functionality of a particular module, the kernel can load that specific module from disk and automatically integrate it into itself, thus dynamically extending its capabilities.

If the features of a loaded kernel module haven't been used for several minutes, the kernel can voluntarily disassociate it from the rest of the kernel and unload it from memory -- something that's called *autocleaning*. Without kernel modules, you'd need to ensure that your running kernel (which exists on disk as a single binary file) contains absolutely all the functionality you could possibly need. Without modules, you'd need to build a completely **new** kernel to add important new functionality to it.

Typically, users build a single kernel image that contains all **essential** functionality, and then build a bunch of modules that correspond to features that they **may** need in the future. If and when that time comes, the appropriate module can be loaded into the kernel as needed. This also helps to conserve RAM, since a module uses RAM only when it has been loaded from

disk. When a module is removed from the kernel, that memory can be freed and used for other purposes.

Where modules live

Kernel modules typically live in /lib/modules/x.y.z (where x.y.z is the kernel version with which the modules are compatible); each module has ".o" at the end of its name, which identifies it as a binary file containing machine instructions. As you may guess, each individual module represents a particular component of kernel functionality. One module may provide FAT filesystem support, while another may support a particular ISA ethernet card.

Modules -- not for every process!

It's worth mentioning that you can't put **everything** in a module. Because modules are stored on disk, your bootable kernel image needs to have compiled-in support for your disk controller, drives, and your root filesystem. If you don't have these essential components compiled into your kernel image -- that is, if you try to compile them as modules instead -- then your kernel won't have the necessary ability to load these modules from disk, creating a rather ugly chicken-and-egg problem that will result in a kernel that can't boot your system!

Section 3. Locating and downloading sources

Kernel version history

At the time this tutorial was written, the most recent kernel available was 2.4.18. The 2.4.18 kernel is part of the 2.4 stable kernel series. This series of kernel releases is intended for production systems.

There are also several 2.5 series kernels available, but you should not use them on production systems. The "5" in "2.5" is an odd number, indicating that these kernels are experimental in nature and intended for kernel developers. When the "2.5" kernels are ready for production use, a "2.6" (even second number) series will begin.

Which kernel sources to use

If you simply want to compile a new version of your currently installed kernel (for example, to enable SMP support), then the best way to proceed is to install your distribution's kernel source package. After doing so, you should find a bunch of new files in /usr/src/linux.

However, there may be times where you want to install a **new** kernel. Generally, the best approach is to simply install a new or updated version of your distribution's kernel source package. This package will contain kernel sources that have been patched and tweaked to run optimally on your Linux system.

Getting the kernel from its source

If you have an adventurous streak, you can grab a "mainline" kernel source tarball from *http://www.kernel.org/pub/linux/kernel* instead. In this directory, you'll find the official kernel sources, as released by Linus or Marcelo. They may not have all the features found in your distribution's kernel source package, so it's generally best to not use a mainline kernel until you feel that you know what you're doing...or until you have an "extra" machine and lots of spare time :)

At kernel.org, you'll find the kernel sources organized into several different directories, based on kernel version (v2.2, v2.4, etc.) Inside each directory, you'll find files labelled "linux-x.y.z.tar.gz" and "linux-x.y.z.tar.bz2." These are the Linux kernel source tarballs. You'll also see files labelled "patch-x.y.z.gz" and "patch-x.y.z.bz2." These files are patches that can be used to update the previous version of complete kernel sources. If you want to compile a new kernel release, you'll need to download one of the "linux" files.

Unpacking the kernel

If you downloaded a new kernel from kernel.org, now it's time to unpack it. To do so, cd into /usr/src. If there is an existing "linux" directory there, move it to "linux.old" (`mv linux linux.old`, as root.)

Now, it's time to extract the new kernel. While still in /usr/src, type `tar xzvf /path/to/my/kernel-x.y.z.tar.gz` or `cat`

`/path/to/my/kernel-x.y.z.tar.bz2 | bzip2 -d | tar xvf -`, depending on whether your sources are compressed with gzip or bzip2. After typing this, your new kernel sources will be extracted into a new "linux" directory. Beware -- the full kernel sources typically occupy more than 50 MB on disk!

Section 4. Configuring the kernel

Let's talk configuration

Before you compile your kernel, you need to configure it. Configuration is your opportunity to control exactly what kernel features are enabled (and which are disabled) in your new kernel. You'll also be in control of what parts get compiled into the kernel binary image (which gets loaded at boot-time), and what parts get compiled into load-on-demand kernel module files.

The old-fashioned way of configuring a kernel was a tremendous pain, and involved entering /usr/src/linux and typing `make config`. While make config still works, please don't try to use this method to configure your kernel -- unless you like answering hundreds (yes, hundreds!) of yes/no questions on the command line.

The new way to configure

Instead of typing "make config," we modern folks type either "make menuconfig" or "make xconfig" to configure our kernels. If you type "make menuconfig," you'll get a nice console-based color menu system that you can use to configure the kernel. If you type "make xconfig," you'll get a very nice X-based GUI that can be used to configure various kernel options.

When using "make menuconfig," options that have a "< >" to their left can be compiled as a module. When the option is highlighted, hit the space bar to toggle whether the option is deselected ("< >"), selected to be compiled into the kernel image ("<*>"), or selected to be compiled as a module ("<M>"). You can also hit "y" to enable an option, "n" to disable it, or "m" to select it to be compiled as a module if possible. Fortunately, most kernel configuration options have verbose help that you can view by typing `h`.

Configuration tips

Unfortunately, there are so many kernel configuration options that we simply don't have room to cover them all here (but you can check the *options(4)* man page for a more complete list of options, if you're curious).

In the following panels, I'll give you an overview of the important categories you'll find when you do a "make menuconfig" or "make xconfig," pointing out essential or important kernel configuration options along the way.

Code maturity level options

Now, let's take a look at the various kernel configuration option categories. I'll include a brief overview of each category below. I encourage you to follow along by typing "make menuconfig" or "make xconfig" in /usr/src/linux.

Code maturity level options: This configuration category contains a single option: "Prompt for development and/or incomplete code/drivers." If enabled, many options that are considered experimental (such as ReiserFS, devfs, and others) will be visible under other

category menus. If this option isn't selected, the only options that will be visible will be those that are considered "stable." Generally, it's a good idea to enable this option so that you can see all that the kernel has to offer.

Modules and CPU-related options

Loadable module support: Under this configuration category are three options related to the kernel's support for modules. In general, all three options should be enabled.

Processor type and features: This section includes various CPU-specific configuration options. Of particular importance is the "Symmetric multiprocessing support option", which should be enabled if your system has more than one CPU. Otherwise, only the first CPU in your system will be utilized. The "MTRR Support" option should generally be enabled, since it will result in better performance in X on modern systems.

General and parallel port options

General setup: In this section, Networking and PCI support options should generally always be enabled, as should "Kernel support for ELF binaries" (build it into the kernel, not as a module). The a.out and MISC binary options are recommended, but generally make more sense as kernel modules. Also be sure to enable "System V IPC" and "Sysctl support." See the built-in help for more information on these options.

Parallel port support option: The **Parallel port support** section should be of interest to anyone with parallel port devices, including printers. Note that in order to have full printer support, you must also enable "Parallel printer support" under the "Character devices" section in addition to the appropriate parallel port support here.

RAID and LVM

Multi-device support (RAID and LVM): This contain options relating to Linux software RAID and logical volume management. Software RAID allows you to use your disks in a redundant fashion in order to increase availability. You can find more information on software RAID in the developerWorks software RAID series (see the final section of this tutorial, "Resources", for links).

Networking and related devices

Networking options: This contains options related to -- you guessed it -- networking! If you're planning to attach your Linux system to a typical network, you should be sure to enable "Packet socket," "Unix domain sockets" and "TCP/IP networking." There are various other options you may be interested in, including "Network packet filtering" that allows you to use the `iptables` command to set up your own stateful firewall. For information on doing this, see the *developerWorks* tutorial *Linux 2.4 stateful firewall design*.

Network device support: The second requirement for getting Linux networking to work is to compile in support for your particular networking hardware. You should select support for the

card(s) that you'd like your kernel to support. The options you want are most likely hiding under the "Ethernet (10 or 100Mbit)" sub-category.

IDE support

ATA/IDE/MFM/RLL support: This section contains important options for those using IDE drives, CD-ROMs, DVD-ROMs, and other peripherals. If your system has IDE disks, be sure to enable "Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support," "Include IDE/ATA-2 DISK support", and the chipset appropriate to your particular motherboard (built-in to the kernel, not as modules -- so your system can boot!). If you have an IDE CD-ROM, be sure to enable "Include IDE/ATAPI CD-ROM support" as well. Note: without specific chipset support, IDE will still work but may not take advantage of all the performance-enhancing features of your particular motherboard.

Also note that the "Enable PCI DMA by default if available" is a highly recommended option for nearly all systems. Without DMA (direct memory access) enabled by default, your IDE peripherals will run in PIO mode and may perform up to **15 times** slower than normal! You can verify that DMA is enabled on a particular disk by typing `hdparm -d 1 /dev/hdx` at the shell prompt as root, where /dev/hdx is the block special device corresponding to the disk on which you'd like to enable DMA.

SCSI support

SCSI support: This contains all the options related to SCSI disks and peripherals. If you have a SCSI-based system, be sure to enable "SCSI support", "SCSI disk support", "SCSI CD-ROM support", and "SCSI tape support" as necessary. If you are booting from a SCSI disk, ensure that both "SCSI support" and "SCSI disk support" are compiled-in to your kernel and are not selected to be compiled as loadable modules. For SCSI to work correctly, you need to perform one additional step: head over to the "SCSI low-level drivers" sub-category and ensure that support for your particular SCSI card is enabled and configured to be compiled directly into the kernel.

Miscellaneous character devices

Character devices: This section contains a potpourri of miscellaneous kernel drivers. Be sure to enable "Virtual terminal" and "Support for console on virtual terminal;" these are needed for the standard text-based console that greets you after the kernel boots. You'll most likely need to enable "Unix98 PTY support" as well. If you want to use a parallel printer, remember to enable "Parallel printer support" too. Everything else is typically optional. "Enhanced real-time clock support" is recommended; "/dev/agpgart (AGP support)" and "Direct Rendering Manager" are typically required to take advantage of free Linux 3D acceleration under X (particularly if you have a Voodoo3+, ATI Rage 128, ATI Radeon, or Matrox card). Getting X to work in accelerated mode requires additional configuration steps besides simply enabling these options.

File systems and console drivers

File systems: This contain options related to filesystem drivers, as you might guess. You'll need to ensure that the filesystem used for "/" (the root directory) is compiled into your kernel. Typically, this is ext2, but it may also be ext3, JFS, XFS, or ReiserFS. Be sure to also enable the "/proc file system support" option, as most distributions require it. Typically, you should also enable "/dev/pts file system support for Unix98 PTYs," unless you're planning to use "/dev file system support," in which case you should leave the "/dev/pts" option disabled.

Console drivers: Typically, most people will enable "VGA text console" (normally required on x86 systems) and optionally "Video mode selection support." It's also possible to use "Frame-buffer support," which will cause your text console to be rendered on a graphics rather than a text screen. Some of these drivers **can** negatively interact with X, so it's best to stick with the VGA text console, at least in the beginning.

Section 5. Compiling and installing the kernel

make dep

Once your kernel is configured, it's time to get it compiled. But before we can compile it, we need to generate dependency information. Do this by typing `make dep` while in /usr/src/linux.

---

make bzImage

Now it's time to compile the actual binary kernel image. Type `make bzImage`. After several minutes, compilation will complete and you'll find the bzImage file in /usr/src/linux/arch/i386/boot (for an x86 PC kernel). You'll see how to install the new kernel image in a bit, but now it's time for the modules.

Compiling modules

Now that the bzImage is done, it's time to compile the modules. Even if you didn't enable any modules when you configured the kernel, don't skip this step -- it's good to get into the habit of compiling modules immediately after a bzImage. And, if you really have **no** modules enabled for compilation -- this step will go really quickly for you. Type `make modules && make modules_install`. This will cause the modules to be compiled and then installed into /usr/lib/<kernelversion>.

Congratulations! Your kernel is now fully compiled, and your modules are all compiled and installed. Now it's time to reconfigure LILO so that you can boot the new kernel.

Section 6. Boot configuration

Intro to LILO

It's finally time to reconfigure LILO so that it loads the new kernel. LILO is the most popular Linux boot loader, and is used by all popular Linux distributions. The first thing you'll want to do is take a look at your /etc/lilo.conf file. It will contain a line that says something like "image=/vmlinuz" This line tells LILO where it should look for the kernel.

Configuring LILO

To configure LILO to boot the new kernel, you have two options. The first is to overwrite your existing kernel -- this is risky unless you have some kind of emergency boot method, such a boot disk with this particular kernel on it.

The safer option is to configure LILO so that it can boot either the new or the old kernel. LILO can be configured to boot the new kernel by default, but still provide a way for you to select your older kernel if you happen to run into problems. This is the recommended option, and the one we'll show you how to perform.

LILO code

Your lilo.conf may look like this:

```
boot=/dev/hda
delay=20
vga=normal
root=/dev/hda1
read-only

image=/vmlinuz
label=linux
```

To add a new boot entry to your lilo.conf, do the following. First, copy /usr/src/linux/arch/i386/boot/bzImage to a file on your root partition, such as /vmlinuz2. Once it's there, duplicate the last three lines of your lilo.conf and add them again to the end of the file... we're almost there...

Tweaking LILO

Now, your lilo.conf should look like this:

```
boot=/dev/hda
delay=20
vga=normal
root=/dev/hda1
read-only

image=/vmlinuz
```

```
label=linux

image=/vmlinuz
label=linux
```

Now, change the first "image=" line to read `image=/vmlinuz2`" Next, change the **second** "label=" line to read `label=oldlinux`. Also, make sure there is a "delay=20" line near the top of the file -- if not, add one. If there is, make sure the number is at least twenty.

The final lilo.conf

Your **final** lilo.conf file will look something like this:

```
boot=/dev/hda
delay=20
vga=normal
root=/dev/hda1
read-only

image=/vmlinuz2
label=linux

image=/vmlinuz
label=oldlinux
```

After doing all this, you'll need to run "lilo" as root. This is very important! If you don't do this, the booting process won't work. Running "lilo" will give it an opportunity to update its boot map.

The whys and wherefores of LILO configuration

Now for an explanation of our changes. This lilo.conf file was set up to allow you to boot two different kernels. It'll allow you to boot your original kernel, located at /vmlinuz. It'll also allow you to boot your new kernel, located at /vmlinuz2. By default, it will try to boot your new kernel (because the image/label lines for the new kernel appear first in the configuration file).

If, for some reason, you need to boot the old kernel, simply reboot your computer and hold down the shift key. LILO will detect this, and allow you to type in the label of the image you'd like to boot. To boot your old kernel, you'd type `oldlinux`, and hit Enter. To see a list of possible labels, you'd hit TAB.

Section 7. PCI devices

PCI devices 101

This section will take a closer look at the finer details of dealing with PCI devices under Linux. Enabling support for PCI devices under Linux is pretty straightforward. Simply ensure that you have "PCI support" enabled under the "General Setup" kernel configuration category. The "PCI device name database" option is also recommended, as it will allow you to view the actual English names of the PCI devices that Linux can see (instead of just their official PCI device ID numbers). Other than ensuring that the above options are enabled, Linux is ready to support PCI.

The only additional step required is to enable the specific driver for the type of card you're installing into your system. For example, you'd enable "Creative SBLive!" support (under the "Sound" category) if you were installing a SoundBlaster Live!" card, and you'd enable "3c590/3c900 series (592/595/597) "Vortex/Boomerang" support" under the "Network device support/Ethernet (10 or 100Mbit)" category/subcategory if you were installing a 3Com 3c905c Fast Ethernet card.

Inspecting your PCI devices

To view information about installed PCI devices, you can type `cat /proc/pci` to view barebones (and somewhat cryptic) information -- or type `lspci -v` for more verbose and understandable output. The "lspci" is part of the pciutils package, whose sources are available from *http://atrey.karlin.mff.cuni.cz/~mj/pciutils.html*. Generally, using the version of pciutils that comes with your particular distribution is sufficient. When you type `lspci -v`, you may see many PCI devices that you never knew even existed. More often than not, such a device is one of the many PCI-based peripheral devices that has been built-in to your computer's motherboard. These devices can be disabled (and enabled if they aren't currently visible) in your computer's BIOS, typically under the "Integrated peripherals" section. You can normally access your computer's BIOS by pressing the Delete key or the F2 key as your system boots.

The pciutils package also contains a program called "setpci" that can be used to change various PCI device settings, including PCI device latency. To learn more about PCI device latency and the effects it can have on your system, see the *developerWorks* article *Linux hardware stability guide, Part 2*.

PCI device resources

In order to do their thing, the PCI devices in your system need to take advantage of various specific hardware resources in your system, such as interrupts. Many PCI devices take advantage of hardware interrupts to signal the processor when they have some data ready for processing. To see what interrupts are being used by your various hardware devices, you can view the /proc/interrupts file by typing `cat /proc/interrupts`. You'll see output that looks something like this:

```
           CPU0
  0:    3493317          XT-PIC  timer
```

```
  1:      86405          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  5:          0          XT-PIC  eth0
  8:          2          XT-PIC  rtc
  9:      62653          XT-PIC  usb-uhci, usb-uhci, eth1
 10:    1550399          XT-PIC  Audigy
 12:     413422          XT-PIC  PS/2 Mouse
 14:      85418          XT-PIC  ide0
 15:          4          XT-PIC  ide1
NMI:          0
ERR:          0
```

The first column lists an IRQ number; the second column displays how many interrupts have been processed by the kernel for this particular IRQ; and the last column identifies the "short name" of the hardware device(s) associated with the IRQ. As you can see, multiple devices are capable of sharing the same IRQ if necessary.

You can also view the IO ports that your hardware devices are using by typing `cat /proc/ioports`.

Section 8. Linux USB

Introducing Linux USB

When configuring the kernel, you probably noticed a "USB support" section containing options pertaining to USB, also known as the Universal Serial Bus. USB is a relatively new way of connecting peripheral devices to PCs. These days, there are USB mice, keyboards, game controllers, printers, modems, and more. Because Linux USB support is relatively new, many Linux users have never used USB devices on their Linux systems, or may not be fully up-to-speed on how Linux USB support works. The following panels will give you a quick introduction to Linux USB to help you get started.

Enabling USB

To enable Linux USB support, first go inside the "USB support" section and enable the "Support for USB" option. While that step is fairly obvious, the following Linux USB setup steps can be confusing. In particular, you now need to select the proper USB host controller driver for your system. Your options are "EHCI," "UHCI," "UHCI (alternate driver)," and "OHCI." This is where a lot of people start getting confused about USB for Linux.

UHCI, OHCI, EHCI -- oh my!

To understand what "EHCI" and friends are, you need to first know that every motherboard or PCI card that includes support for plugging in USB devices needs to have a USB host controller chipset on it. This particular chipset interfaces with the USB devices that you plug in to your system and takes care of all the low-level details necessary to allow your USB devices to communicate with the rest of the system.

The Linux USB drivers have three different USB host controller options because there are three different types of USB chips found on motherboards and PCI cards. The "EHCI" driver is designed to provide support for chips that implement the new high-speed USB 2.0 protocol. The "OHCI" driver is used to provide support for USB chips found on non-PC systems, as well as those on PC motherboards with SiS and ALi chipsets. The "UHCI" driver is used to provide support for the USB implementations you'll find on most other PC motherboards, including those from Intel and Via. You simply need to select the "?HCI" driver that corresponds to the type of USB support you'd like to enable. If in doubt, you can enable "ECHI," "UHCI" (pick either of the two, there's no significant difference between them), and "OHCI" just to be safe.

The last few steps

Once you've enabled "USB support" and the proper "?HCI" USB host controller drivers, there are just a few more steps involved in getting USB up and running. You should enable the "Preliminary USB device filesystem," and then ensure that you enable any drivers specific to the actual USB peripherals that you will be using with Linux. For example, in order to enable support for my USB game controller, I enabled the "USB Human Interface Device (full HID) support". I also enabled "Input core support" and "Joystick support" under the main "Input core support" section.

Mounting usbdevfs

Once you've rebooted with your new USB-enabled kernel, you should mount the USB device filesystem to /proc/bus/usb by typing the following command:

```
# mount -t usbdevfs none /proc/bus/usb
```

In order to have the USB device filesystem mounted automatically when your system boots, add the following line to /etc/fstab after the /proc mount line:

```
none            /proc/bus/usb            usbdevfs        defaults   0   0
```

For more information about USB, visit the USB sites I've listed in "Resources," which follow.

Section 9. Resources and feedback

Resources

The Linux Kernel HOWTO is another good resource for kernel compilation instructions.

The LILO, Linux Crash Rescue HOW-TO shows you how to create an emergency Linux boot disk.

www.kernel.org hosts the Linux Kernel archives.

Don't forget *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 Administrator's 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.

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 series (particularly parts 1 and 2) are excellent additional preparation for the LPI exam:

- *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*

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

If you're not too familiar with the `vi` editor, you should check out Daniel's *tutorial on Vi*. This *developerWorks* 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`.

You can find more information on software RAID in Daniel's *developerWorks* software RAID series: *Part 1* and *Part 2*. Logical volume management adds an additional storage management layer to the kernel that allows you to easily grow, shrink, and span filesystems across multiple disks. To learn more about LVM, see Daniel's articles on the subject: *Part 1* and *Part 2*. Both software RAID and LVM require additional user-land tools and setup.

For more information on using the `iptables` command to set up your own stateful firewall, see the *developerWorks* tutorial *Linux 2.4 stateful firewall design*.

For more information about USB, visit *http://www.linux-usb.org/* . For additional USB setup and configuration instructions, be sure to read the *Linux-USB guide* .

For more information on the Linux Professional Institute, visit the *LPI home page*.

Your feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact the author, 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.

LPI certification 102 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 "Networking," the third of four tutorials designed to prepare you for the Linux Professional Institute's 102 exam. In this tutorial, we'll introduce you to TCP/IP and Ethernet Linux networking fundamentals, show you how to use the inetd and xinetd superservers, provide you with important tips for securing your Linux systems, and also show you how to set up and use a Linux print server. By the end of this series of tutorials (eight in all; this is part seven), you'll 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 the *Linux Professional Institute*.

Should I take this tutorial?

This tutorial is ideal for those who want to learn about or improve their basic Linux networking and security skills. It's especially appropriate for those who will be setting up applications on Linux servers or desktops. For many, 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 important Linux system administration skills.

If you are new to Linux, we recommend that you first complete the previous tutorials in the LPI certification 101 and 102 exam prep series before continuing:

- *101 series, Part 1: Linux fundamentals*
- *101 series, Part 2: Basic administration*
- *101 series, Part 3: Intermediate administration*
- *101 series, Part 4: Advanced administration*
- *102 series, Part 1: Compiling sources and managing packages*
- *102 series, Part 2: Compiling and configuring the kernel*

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@gentooorg*

Residing in Albuquerque, New Mexico, Daniel Robbins 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 to 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 he 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 to be 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, New Hampshire.

Section 2. TCP/IP networking

Introduction

Setting up an Ethernet-based Local Area Network (LAN) consisting of a bunch of Linux machines is a common and relatively simple task. Generally, all you need to do is make sure that your Linux systems have an Ethernet card of some kind installed in them. Then, connect the machines to a central Ethernet hub or switch using Ethernet cabling. If all your systems have support for their respective Ethernet card compiled into the kernel (as well as TCP/IP support), then they technically have everything they need to communicate over your new Ethernet LAN.

Ethernet alone isn't much fun

While you'll then have all the hardware and kernel support needed for your LAN to work, it won't do much. The vast majority of Linux applications and services don't exchange information using raw Ethernet packets, or *frames*. Instead, they use a higher-level protocol called TCP/IP. You've undoubtedly heard of TCP/IP -- it's the suite of protocols that forms the foundation of the Internet in general (hence the name, Transmission Control Protocol/Internet Protocol).

The solution: TCP/IP over Ethernet

The solution, then, is to configure your new Ethernet LAN so that it can exchange TCP/IP traffic. To understand how this works, we first need to understand a bit about Ethernet. On an Ethernet LAN, in particular, the Ethernet card in every machine has a unique hardware address. This hardware address is assigned to the card at the time of manufacture, and looks something like this:

```
00:01:02:CB:57:3C
```

Introducing IP addresses

These hardware addresses are used as unique addresses for individual systems on your Ethernet LAN. Using hardware addresses, one machine can, for example, send an Ethernet frame addressed to another machine. The problem with this approach is that TCP/IP-based communication uses a different kind of addressing scheme, using what are called IP addresses instead. IP addresses look something like this:

```
192.168.1.1
```

Associating an IP address with an Ethernet interface

In order to get your Ethernet LAN working for TCP/IP, you need some way of associating each machine's Ethernet card (and thus its hardware address) with an IP address. Fortunately, there's an easy way to associate an IP address with an Ethernet interface under

Linux. In fact, if you are currently using Ethernet with Linux, your distribution's system initialization scripts very likely have a command in them that looks something like this:

```
ifconfig eth0 192.168.1.1 broadcast 192.168.1.255 netmask 255.255.255.0
```

Above, the `ifconfig` command is used to associate eth0 (and thus eth0's hardware address) with the 192.168.1.1 IP address. In addition, various other IP-related information is specified, including a broadcast address (192.168.1.255) and a netmask (255.255.255.0). When this command completes, your eth0 interface will be enabled and have an associated IP address.

Using ifconfig -a

You can view all network devices that are currently running by typing `ifconfig -a`, resulting in output that looks something like this:

```
eth0      Link encap:Ethernet  HWaddr 00:01:02:CB:57:3C
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:5 Base address:0xc400

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:1065 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1065 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:262542 (256.3 Kb)  TX bytes:262542 (256.3 Kb)
```

Above, you can see a configured eth0 interface, as well as a configured lo (localhost) interface. The lo interface is a special virtual interface that's configured so that you can run TCP/IP applications locally, even without a network.

TCP/IP is working!

Once all of your network interfaces are brought up and associated with corresponding IP addresses, your Ethernet network can be used to carry TCP/IP traffic as well. The systems on your LAN can now address each other using IP addresses, and common commands such as `ping`, `telnet`, and `ssh` will work properly between your machines.

Name resolution limitations

However, while you'll be able to type things like `ping 192.168.1.1`, you won't be able to refer to your boxes by name. For example, you won't be able to type `ping mybox`. To do this, you need to set up a file called /etc/hosts on each of your Linux boxes. In this file, you specify an IP address, along with the name (or names) that are associated with each IP

address. So if I had a network with three nodes, my /etc/hosts file might look something like this:

```
127.0.0.1                                   localhost
192.168.1.1                                   mybox mybox.gentoo.org
192.168.1.2                                   testbox testbox.gentoo.org
192.168.1.3                              mailbox mailbox.gentoo.org
```

Note that /etc/hosts contains an obligatory mapping of "localhost" to the 127.0.0.1 IP address. I've also specified the hostnames of all the systems on my LAN, including both their short name ("mybox") and their fully-qualified name ("mybox.gentoo.org"). After copying this /etc/hosts file to each of my systems, I'll now be able to refer to my systems by name, rather than simply by IP address: `ping mybox` will now work!

Using DNS

While this approach works for small LANs, it isn't very convenient for larger LANs with many systems on them. For such configurations, it's generally much better to store all your IP-to-hostname mapping information on a single machine, and set up what is called a "DNS server" (domain name service server) on it. Then, you can configure each machine to contact this particular machine to receive up-to-the-minute IP to name mappings. This is done by creating an /etc/resolv.conf file on every machine that looks something like this:

```
domain gentoo.org
nameserver 192.168.1.1
nameserver 192.168.1.2
```

In the above /etc/resolv.conf, I tell the system that any host names that are not qualified (such as "testbox" as opposed to "testbox.gentoo.org," etc.) should be considered to be *local* hostnames. I also specify that I have a DNS server running on 192.168.1.1, as well as a backup one running on 192.168.12. Actually, nearly all network-connected Linux PCs already have a nameserver specified in their resolv.conf file, even if they aren't on a LAN. This is because they are configured to use a DNS server at their Internet Service Provider, in order to map hostnames to IP addresses (so that users on that system can do things like browse the Web and head over to well-known sites like ibm.com without having to refer to them by IP address!).

Connecting to the outside

Speaking of connecting to the Internet, how would we configure our simple 3-system LAN so that it connect to systems on the "outside?" Typically, we'd purchase a router of some kind that can connect to both our Ethernet network *and* a DSL or Cable modem, or a T1 or phone line. This router would be configured with an IP address so that it could communicate with the systems on our LAN. In turn, we would configure every system on our LAN to use this router as its *default route*, or gateway. What this means is that any network data addressed to a system that *isn't* on our LAN would be routed to our router, which would take care of forwarding to remote systems outside our LAN. Generally, your distribution's system initialization scripts handle the setting of a default route for you. The command they use to do this probably looks something like this:

```
route add -net default gw 192.168.1.80 netmask 0.0.0.0 metric 1
```

In the above `route` command, the default route is set to 192168.1.80 -- the IP address of the router. To view all the routes configured on your system, you can type `route -n`. The route with a destination of "0.0.0.0" is the default route.

Homework

So far, we've given you a very brief introduction to Linux networking concepts. Unfortunately, we simply don't have the room to cover everything you need to know, such as how to select appropriate IP addresses, network masks, broadcast addresses, etc. In fact, there's quite a bit more information you'll need to learn in order to prepare for the LPI Level 102 exam.

Fortunately, the topic of Linux networking is one of the most comprehensively documented Linux topics around. In particular, we recommend that you read the *Linux Network Administrators Guide*, available from *Linuxdoc.org's "Guides" section*, especially sections 2 through 6. Accompanied with our gentle introduction to Linux networking, the Linux Network Administrators Guide should get you up to speed in no time!

Section 3. Internet services

Introducing inetd

A single Linux system can provide dozens, even hundreds, of network services. For example, when you use the telnet program, you are accessing the telnet service on a remote system. Likewise, when you use the ftp program, you are connecting to the ftp service on the remote system.

In order to provide these services, the remote system either runs an instance of each server to accept connections (for example /usr/sbin/in.telnetd and /usr/sbin/in.ftpd), or runs inetd. The inetd program accepts each incoming connection and starts the appropriate services to handle the connection based on its type. For this reason, inetd is also known as the "Internet superserver."

On a typical Linux installation, inetd handles most incoming connections. Only a few programs (such as sshd and lpd) handle their own network communication without relying on inetd to accept incoming connections.

Configuring inetd: /etc/services

The previous panel mentioned that inetd classifies incoming connections based on type. Each incoming connection includes some identification fields in the TCP/IP header. The fields that interest us the most are source address, destination address, protocol, and port number. Incoming connections are classified by inetd based on port number and protocol (usually TCP or UDP, see /etc/protocols for the complete list of services that can be serviced by inetd).

Each line has the format:

```
service-name  port-number/protocol-name  aliases   # comment
```

For example, let's investigate the top few entries:

```
# grep ^[^#] /etc/services | head -5
tcpmux    1/tcp              # TCP port service multiplexer
echo      7/tcp
echo      7/udp
discard   9/tcp   sink null
discard   9/udp   sink null
```

In general, /etc/services already contains all the useful service names and ports. If you wish to add your own, you should first consult the list of *assigned port numbers*.

Configuring inetd; /etc/inetd.conf

The actual configuration of inetd is done in /etc/inetd.conf, which has the following format:

```
service-name socket-type protocol wait-flag user server-program
```

Since services are specified in inetd.conf by *service name* rather than port, they must be listed in /etc/services in order to be eligible for handling by inetd.

Let's look at some common lines from /etc/inetd.conf. For example, the telnet and ftp services:

```
# grep ^telnet /etc/inetd.conf
telnet  stream  tcp     nowait  root    /usr/sbin/in.telnetd

# grep ^ftp /etc/inetd.conf
ftp     stream  tcp     nowait  root    /usr/sbin/in.ftpd -l -a
```

For both of these services, the configuration is to use the TCP protocol, and run the server (`in.telnetd` or `in.ftpd`) as the root user. For a complete explanation of the fields in /etc/inetd.conf, see the *inetd(8)* man page.

Disabling services

Disabling a service in inetd is simple: Just comment out the line in /etc/inetd.conf. You probably don't want to remove the line entirely, just in case you need to reference it later. For example, some system administrators prefer to disable telnet for security reasons (since the connection is entirely cleartext):

```
# vi /etc/inetd.conf
[comment out undesired line]

# grep ^.telnet /etc/inetd.conf
#telnet  stream  tcp     nowait  root    /usr/sbin/in.telnetd
```

Stopping/starting inetd using an init-script

The changes to /etc/inetd.conf that we made in the previous panel won't take effect until we restart the inetd program. Most distributions have an init-script in /etc/init.d or in /etc/rc.d/init.d:

```
# /etc/rc.d/init.d/inet stop
Stopping INET services:                    [  OK  ]

# /etc/rc.d/init.d/inet start
Starting INET services:                    [  OK  ]
```

In fact, you can usually use "restart" as a shortcut:

```
# /etc/rc.d/init.d/inet restart
Stopping INET services:                    [  OK  ]
Starting INET services:                    [  OK  ]
```

Stopping/starting inetd manually

If the helper script in the previous panel doesn't work for you, the old-fashioned method is even easier. You can stop inetd using the `killall` command:

```
# killall inetd
```

You can start it again simply by invoking it on the command line. It will automatically run in the background:

```
# /usr/sbin/inetd
```

And there's a shortcut to instruct inetd to reread the configuration file without actually stopping it: just send it the HUP signal:

```
# killall -HUP inetd
```

At this point you shouldn't be able to telnet or ftp into this system, since telnet and ftp are disabled. Try `telnet localhost` to check. If you need telnet or ftp access, all you need to do is to re-enable it!

Here's what happens for me:

```
# telnet localhost
telnet: Unable to connect to remote host: Connection refused
```

Introducing TCP wrappers

The tcp_wrappers package provides a tiny daemon called tcpd that is called by inetd instead of by the actual service daemon. The tcpd program logs the source address of each incoming connection, and can also filter them to allow connections only from trusted systems.

To use tcpd, you can insert it into your inetd as follows:

```
ftp     stream  tcp     nowait  root    /usr/sbin/tcpd  in.ftpd -l -a
telnet  stream  tcp     nowait  root    /usr/sbin/tcpd  in.telnetd
```

Logging with TCP wrappers

By default, connections are unrestricted but are logged. For example, we can restart inetd so that the changes from the previous panel take effect. Then some quick investigation should show the logged connections:

```
# telnet localhost
login: (press <ctrl-d> to abort)
```

```
# tail -1 /var/log/secure
Feb 12 23:33:05 firewall in.telnetd[440]: connect from 127.0.0.1
```

The telnet attempt was logged by tcpd, so it looks like we have things working. Since tcpd provides a consistent connection logging service, that frees up the individual service daemons from each needing to log connections on their own. In fact, it's similar to inetd doing the work of accepting connections, since that frees up each of the individual daemons from needing to accept their own connections. Isn't the simplicity of Linux (UNIX) marvelous?

Restricting access to local users with TCP wrappers

The tcpd program is configured using two files: /etc/hosts.allow and /etc/hosts.deny. These files have lines of the form:

```
daemon_list : client_list [ : shell_command ]
```

Access is granted or denied in the following order. The search stops at the first match:

- Access is granted when a match is found in **/etc/hosts.allow**
- Access is denied when a match is found in **/etc/hosts.deny**
- Access is granted if nothing matches

For example, to allow telnet access only to our internal network, we start by setting policy (reject all connections with a source other than localhost) in /etc/hosts.deny:

```
in.telnetd: ALL EXCEPT LOCAL
```

Restricting access to known hosts with TCP wrappers

There's no need to reload inetd, since tcpd is invoked each time there's a connection on the telnet port. So we can try it immediately:

```
# telnet box.yourdomain.com
Trying 10.0.0.1...
Connected to box.yourdomain.com.
Escape character is '^]'.
Connection closed by foreign host.
```

Slap! Rejected! (This is one of the few times in life that rejection is indicative of success.) To re-enable access from our own network, we insert the exception in /etc/hosts.allow:

```
in.telnetd: .yourdomain.com
```

At this point we're able to successfully telnet into our system again. This is just scraping the surface of the capabilities of tcp_wrappers. There's lots more information on tcp_wrappers in the *tcpd(8)* and *hosts_access(5)* man pages.

xinetd: inetd extended

Although inetd is the classic Internet superserver, there have been recent rewrites that attempt to add features and more security. The xinetd program replaces inetd in many modern distributions, including Red Hat and Debian. Some of its extended features are:

- Access control (built-in TCP wrappers)
- Extensive logging (connection durations, failed connections, etc.)
- Redirection of services from another host
- IPv6 support
- Configuration via snippets rather than one consolidated file

xinetd configuration

The configuration file for xinetd is /etc/xinetd.conf. Most often, that file contains just a few lines that set default configuration parameters for the rest of the services:

```
# cat /etc/xinetd.conf
defaults
{
    instances      = 60
    log_type       = SYSLOG authpriv
    log_on_success = HOST PID
    log_on_failure = HOST RECORD
}
includedir /etc/xinetd.d
```

The last line in that file instructs xinetd to read additional configuration from file snippets in the /etc/xinetd.d directory. Let's take a quick glance at the telnet snippet:

```
# cat /etc/xinetd.d/telnet
service telnet
{
    flags          = REUSE
    socket_type    = stream
    wait           = no
    user           = root
    server         = /usr/sbin/in.telnetd
    log_on_failure += USERID
}
```

As you can see, it's not hard to configure xinetd, and it's more intuitive than inetd. You can get lots more information about xinetd in the *xinetd(8)*, *xinetd.conf(5)*, and *xinetd.log(5)* man pages.

There's also lots of information on the Web regarding inetd, tcp_wrappers, and xinetd. Be sure to check out some of the links we've provided for these tools in the last section of this tutorial, Resources; they will give you a much better feel for the capability and configuration of these tools.

Section 4. Security overview

Introduction

Maintaining a completely secure system is impossible. With diligence, however, it is possible to keep your Linux machines secure enough that most casual crackers, script-kiddies, and other Bad Guys will be thwarted and go bug someone else. Remember that simply following this tutorial will not result in a secure system. Instead, we hope to touch on many of the major topic areas and gives you some useful examples of how to get started.

Linux system security can be divided into two parts: internal security and external security. *Internal security* refers to guarding against users accidentally or maliciously disrupting the system. *External security* refers to preventing unauthorized users from gaining access to the system.

This section will cover internal security first, then external security, and we'll finish with some general guidelines and tips.

File permissions for log files

Internal security can be a large task, depending on how much you are able to trust your users. The guidelines presented here are designed to prevent the casual user from accessing sensitive information and from unfairly using system resources.

Regarding file permissions, you may want to modify permissions for the following three cases:

First, log files in /var/log need not be world-readable. There is no reason for anybody other than root to be snooping in the logs. See *Part 4 of the LPI 101 series* for more information on syslog, plus the *logrotate(8)* man page for information on configuring that program to create logs with appropriate permissions.

File permissions for root's other files

Second, root's dot-files shouldn't be readable by an ordinary user. Check the files in root's home directory (`ls -la`) to make sure they're appropriately protected. You can even make the whole directory readable only by root:

```
# cd
# pwd
/root
# chmod 700 .
```

File permissions for user files

Finally, user files are often created world-readable by default. That probably isn't the expectation of your users, and it certainly isn't the best policy. You should set the default umask in /etc/profile using something like the following:

```
if [ "$UID" = 0 ]; then
  # root user; set world-readable by default so that
  # installed files can be read by normal users.
  umask 022
else
  # make user files secure unless they explicitly open them
  # for reading by other users
  umask 077
fi
```

You should consult the *umask(2)* and *bash(1)* man pages for more information on setting the umask. Note that the *umask(2)* man page refers to the C function, but the information it contains applies to the bash command as well. See *Part 3 of the LPI 101 series* for additional details on umask.

Finding SUID/SGID programs

A malicious user seeking root access will always look for programs on the system that have the SUID or SGID bit set. As we discussed in *Part 3 of the LPI 101 series*, these bits cause the program to always run as the user or group that owns the file. Sometimes this is required for proper functioning of the program. The problem is that *any* program may contain a bug that would allow the user to gain privileges if the program is used improperly.

You should consider each program carefully to determine if it needs to have its SUID or SGID bits on. There may be SUID/SGID programs on your system that you don't need at all.

To search for programs of this nature, use the `find` command. For example, we could start searching for SUID/SGID programs in the /usr directory:

```
# cd /usr
# find . -type f -perm +6000 -xdev -exec ls {} \;
-rwsr-sr-x    1 root     root       593972 11-09 12:47 ./bin/gpg
-r-xr-sr-x    1 root     man         38460 01-27 22:13 ./bin/man
-rwsr-xr-x    1 root     root        15576 09-29 22:51 ./bin/rcp
-rwsr-xr-x    1 root     root         8256 09-29 22:51 ./bin/rsh
-rwsr-xr-x    1 root     root        29520 01-17 19:42 ./bin/chfn
-rwsr-xr-x    1 root     root        27500 01-17 19:42 ./bin/chsh
-rwsr-xr-x    1 lp       root         8812 01-15 23:21 ./bin/lppasswd
-rwsr-x---    1 root     cron        10476 01-15 22:16 ./bin/crontab
```

In this list, I've already found a candidate for closer inspection: lppasswd is part of the CUPS printing software distribution. Since I don't provide print services on my system, I might consider removing CUPS, which will also remove the lppasswd program. There may be no security-compromising bugs in lppasswd, but why take the chance on a program I'm not using? Similarly, *all* services that you don't use should be turned off. You can always enable them if and when you need them.

Setting user limits with ulimit

The ulimit command in bash provides a method for limiting the usage of resources by a given user. Once a limit is lowered, there is no way to raise the limit for the life of the process. Furthermore, the limit is inherited by all child processes. The effect is that you can call ulimit

in /etc/profile, and the limits will irrevocably apply to all users (assuming they're running bash or another shell that runs /etc/profile on login).

To retrieve the current limits, use `ulimit -a`:

```
# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
stack size              (kbytes, -s) unlimited
cpu time               (seconds, -t) unlimited
max user processes              (-u) 3071
virtual memory          (kbytes, -v) unlimited
```

It can be quite tricky to set these limits in a way that actually increases the security of your system without causing problems for legitimate users, so be careful when adjusting these settings.

Setting CPU time limits with ulimit

As an example of ulimit, let's try setting the CPU time for a process to 1 second, then make it timeout with a busy loop. Make sure to start a new bash process (as we do below) in which to try it; otherwise you'll be logged out!

```
# time bash
# ulimit -t 1
# while true; do true; done
Killed

real    0m28.941s
user    0m1.990s
sys     0m0.017s
```

In the example above, "user" time plus "sys" time equals total CPU time used by the process. When the bash process reached the 2-second mark, Linux judged that it had exceeded the 1-second limit, so the process was killed. Cool, eh?

Note: One second was just an example. Don't do this to your users! Even multiple hours is bad, since X can really rack up the time (my current session has used 69+ hours of CPU time). For a real implementation, you may want to ulimit something other than CPU time.

User limits, continued

You may also want to limit things such as the number of simultaneous logins or disk usage. These aren't covered by ulimit; instead you should look into one of the following packages:

- *Clobberd* monitors user activity, and meters resources such as time and network activity.
- *Idled* can log out users that have been idle for too long or who have been logged on for too

long. It can also prevent users from being logged in too many times, and refuse users from being logged in at all.
- *Part 4 of the LPI 101 series* discusses the implementation of filesystem quotas.

Intrusion prevention

External security can be split into two categories: intrusion prevention and intrusion detection. Intrusion prevention measures are taken to prevent unauthorized access to a system. If these measures fail, intrusion detection may prove useful in determining when unauthorized access has occurred, and what damage has been done.

A full Linux installation is a large and complex system. It's difficult to keep track of everything that's installed, and even harder to configure each package's security features. The problem becomes simpler when fewer packages are installed. A first step to intrusion prevention is to remove packages you don't need. Take a look back at *Part 4 of the LPI 101 series* for a review of packaging systems.

Turning off unused network services (superserver)

Turning off unused network services is always a good way to improve your intrusion prevention. For example, if you are running an Internet superserver (such as inetd or xinetd described earlier in this tutorial), then in.rshd, in.rlogind, and in.telnetd are often enabled by default. These network services have nearly all been superseded by more secure alternatives such as ssh.

To disable services in inetd, simply comment out the appropriate line in /etc/inetd.conf by prepending "#;" then restart inetd. (This was described previously in this tutorial, so glance back a few panels if you need a refresher.)

To disable services in xinetd, you can do something similar with the appropriate snippet in /etc/xinetd.d. For example, to disable telnet, either comment out the entire content of the file /etc/xinetd.d/telnet, or simply delete the file. Restart xinetd to complete the procedure.

If you're using tcpd in conjunction with inetd, or if you're using xinetd, you also have the option of limiting incoming connections to trusted hosts. For tcpd, see the earlier sections in this tutorial. For xinetd, search for "only_from" in the *xinetd.conf(5)* man page.

Turning off unused network services (standalone servers)

Some servers are not launched by inetd or xinetd, but are instead running all the time as "standalone" servers. This often includes servers such as atd, lpd, sshd, nfsd, and others. In fact, inetd and xinetd are both standalone servers themselves, and if you have commented out all of the services in their respective config files, you may choose to turn them off completely.

Standalone servers are usually started by the init system when the system boots up or changes runlevels. If you don't remember how runlevels work, take a look at *Part 4 of the LPI*

101 series.

To stop the init system from starting a server, find the symlinks to its startup script in each runlevel directory, and delete them. The runlevel directories are usually named /etc/rc3.d or /etc/rc.d/rc3.d (for runlevel 3). You'll also want to check the other runlevels.

Once the runlevel symlinks for the service are removed, you will still need to shut down the currently running server. It is best to do this with the service's init script, usually found in /etc/init.d or /etc/rc.d/init.d. For example, to shut down sshd:

```
# /etc/init.d/sshd stop
* Stopping sshd... [ ok ]
```

Testing your changes

After you've modified your inetd or xinetd configuration to disable or restrict services, or to shut down a server with its init script, you should test your changes. You can test tcp ports using the telnet client by specifying the service name or number. For example, to verify that rlogin has been disabled:

```
# grep ^login /etc/services
login           513/tcp
# telnet localhost 513
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
```

In addition to the standard telnet client, you should look into the possibility of using utilities for testing the "openness" of your system. We recommend netcat and nmap.

ncat is the network Swiss Army knife: it is a simple UNIX utility that reads and writes data across network connections, using TCP or UDP protocol. **nmap**is a utility for network exploration or security auditing. Specifically, nmap scans ports to determine what's open.

You'll find links to these utilities in the last section of this tutorial, Resources.

Refusing logins for maintenance

In addition to the above methods, there is a generic way to refuse logins by creating the file /etc/nologin. Usually this method is used for short-term maintenance operations. Logins by root will still be allowed, but logins by other users will be denied. For example:

```
# cat > /etc/nologin
==============================================

  System is currently undergoing maintenance
  until 2:00.  Please come back later.

==============================================
# telnet localhost
login: agriffis
```

```
Password:
==========================================

  System is currently undergoing maintenance
  until 2:00.  Please come back later.

==========================================

Login incorrect
```

Be sure to delete the file when you're done with maintenance, otherwise nobody will be able to login until you remember! Not that I've ever done this, no, not me... ;-)

Introducing iptables (ipchains)

The `iptables` and `ipchains` commands are used to adjust and inspect the network packet filter rules in a running Linux kernel. The `ipchains` command was used for 2.2.x versions of the kernel, and although it can still be used with 2.4.x kernels, it has been superseded by `iptables`.

The packet filter rules can be set up to do both firewall and router activities. You can inspect your current rules with the `-L` option to `iptables`:

```
# iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

This is an example of a wide-open system, with no routing or firewalling enabled.

iptables and Linux packet filter

Using the Linux packet filter effectively requires a solid understanding of TCP/IP networking and how it is implemented in the Linux kernel. The netfilter home page (see Resources, the last section of this tutorial, for a link) is a good place to learn more.

Until you're comfortable building your own ruleset, there are many scripts out there that can get you started, as long as you trust their authors. One of the most complete is **gShield** (see Resources). You can adjust its well-commented and fairly simple configuration file to set up most normal forms of packet filter rules.

Intrusion detection - syslogs

Intrusion detection is often neglected by system administrators who trust in the intrusion prevention devices they have in place. Unfortunately, this means that when a hacker finds a crack through which to crawl, the system may be under their control for a long period of time

before their presence is noticed.

The most basic form of intrusion detection is to pay attention to your syslogs. These usually appear in the /var/log directory, although the actual filenames will vary depending on your distribution and configuration.

```
# less /var/log/messages
Feb 17 21:21:38 [kernel]   Vendor: SONY      Model: CD-RW  CRX140E    Rev: 1.0n
Feb 17 21:21:39 [kernel] eth0: generic NE2100 found at 0xe800, Version 0x031243,
DMA 3 (autodetected), IRQ 11 (autodetected).
Feb 17 21:21:39 [kernel] ne.c:v1.10 9/23/94 Donald Becker (becker@scyld.com)
Feb 17 21:22:11 [kernel] NVRM: AGPGART: VIA MVP3 chipset
Feb 17 21:22:11 [kernel] NVRM: AGPGART: allocated 16 pages
Feb 17 22:20:05 [PAM_pwdb] authentication failure; (uid=1000)
  -> root for su service
Feb 17 22:20:06 [su] pam_authenticate: Authentication failure
Feb 17 22:20:06 [su] - pts/3 chouser-root
```

It can take some practice to understand all these messages, but most of the important ones are fairly clear. For example, at the end of this log we can see that user "chouser" tried to use `su` to become root and failed.

Intrusion detection - tripwire

There are a number of packages available that can take a "snapshot" of your whole filesystem and compare it to earlier snapshots to see what has changed. If you can clearly define which files *should* change as part of the normal operation of your system, these packages can very quickly alert you to the presence and activity of a hacker.

Tripwire is one of the most popular of these intrusion detection packages (see Resources at the end of this tutorial for a link). Once you have installed tripwire, you must customize its configuration file so that it knows which files should change and which should not. You will also need to tell it how to send you reports of what has changed, and how often it should run (usually once per day).

Intrusion detection - portsentry

The **PortSentry** package from Psionic Technologies is actually a bit of a cross between intrusion prevention and detection. It watches your network connection, and if it sees any attempts to connect to your system that it deems "suspicious," it will log the event and then block it from happening again. It, too, can be found in Resources at the end of this tutorial.

When you have it installed and running, you will be able to see any attempted connections and how PortSentry responded to them in your syslog:

```
# tail /var/log/messages
Oct 15 00:21:24 mycroft portsentry[603]: attackalert:
  SYN/Normal scan from host: 302.174.40.34/302.174.40.34 to TCP port: 111
Oct 15 00:21:24 mycroft portsentry[603]: attackalert:
  Host 302.174.40.34  has  been  blocked  via wrappers with string:
  "ALL: 302.174.40.34"
```

```
Oct 15 00:21:24 mycroft portsentry[603]: attackalert:
  Host 302.174.40.34 has been blocked via dropped route using command:
  "/sbin/route add -host 302.174.40.34 reject"
Oct 15 00:21:24 mycroft portsentry[603]: attackalert:
  SYN/Normal scan from host: 302.174.40.34/302.174.40.34 to TCP port: 111
Oct 15 00:21:24 mycroft portsentry[603]: attackalert:
  Host: 302.174.40.34/302.174.40.34 is already blocked Ignoring
Oct 15 00:33:59 mycroft portsentry[603]: attackalert:
  SYN/Normal scan from host: 302.106.103.19/302.106.103.19 to TCP port: 111
Oct 15 00:33:59 mycroft portsentry[603]: attackalert:
  Host 302.106.103.19 has been blocked via wrappers with string:
  "ALL: 302.106.103.19"
Oct 15 00:33:59 mycroft portsentry[603]: attackalert:
  Host 302.106.103.19 has been blocked via dropped route using command:
  "/sbin/route add -host 302.106.103.19 reject"
```

General guidelines: keeping software up to date

Since all software has the potential for security holes, it's important to install security fixes for packages as soon as they become available. This is the single most often offered piece of advice by security experts, and the single most often ignored piece of advice by novice administrators. Don't learn this lesson the hard way -- when your boxes have been rooted by a years-old backdoor because you neglected to keep patches up to date.

The debate over whether open source or closed-source software is more secure is a hotly debated one. The best conclusion that can be drawn to date is that they are both usually adequate *when properly administered*, which includes keeping security patches up to date!

Several Web sites can help you keep your software up to date, and keep you aware of known threats. These include the security-conscious *CERT* and *SecurityFocus' BugTraq list*, as well as your normal software update sites like *freshmeat.net* and your distribution's home page. We'll repeat these URLs also in Resources, but security is such an important issue that -- if you are not already familiar with these sites, we do recommend that you take a few minutes to visit the first two now.

General guidelines: high-quality passwords

As mundane as it may sound, choosing high-quality passwords for yourself, and "encouraging" (that is, mandating that) your users to do the same, is a cornerstone of good security. Remember to avoid common words and names, especially any that are related to you such as the name of a friend, where you work, or your pet's name. Also avoid guessable numbers such as your birthday or anniversary. Instead try to use random combinations of letters, numbers, and punctuation.

General guidelines: testing your security

Testing the security of your system is important, but don't let a successful test lull you into a false sense of security. Just because these testing tools don't find a hole is no guarantee that someone with knowledge and imagination -- and a whole lot of time on their hands -- will also fail.

We already mentioned nmap and netcat for testing network security. It is also a good idea to check for weak passwords, especially if your system has multiple users. There are many tools available, such as the ones we've put into Resources at the end of this tutorial.

Section 5. Printing

Introduction

This section will walk you through the set-up and use of the classic UNIX printing system on Linux, sometimes called Berkeley LPD. There are other systems available for Linux that are not covered here; see the Resources section at the end of the tutorial for information on these.

Physically installing a printer is beyond the scope of this tutorial. Once it's correctly hooked up, you'll want to install a print spooler daemon so that machines on the network (including the one housing the spooler) can send print jobs to the printer.

Installing a print spooler daemon (lpd)

One of the best LPD print spoolers is *LPRng*. The method of installation depends on your distribution; see *Part 1 of the LPI 102 series* for details on installing software packages in either Red Hat or Debian.

Once it's installed, the print spooler daemon (officially the Line Printer Daemon) can be run from the command line. Log in as a normal user and try:

```
$ /usr/sbin/lpd --help
--X option form illegal
usage: lpd [-FV] [-D dbg] [-L log]
 Options
 -D dbg      - set debug level and flags
                 Example: -D10,remote=5
                 set debug level to 10, remote flag = 5
 -F          - run in foreground, log to STDERR
               Example: -D10,remote=5
 -L logfile  - append log information to logfile
 -V          - show version info
```

Now that the daemon is installed, you should make sure that it's set up to run automatically. Your distribution's LPRng package may have set this up for you already, but if not, see *Part 4 of the LPI 101 series* for information on using runlevels to start daemons such as lpd automatically.

Basic printer settings (/etc/printcap)

The print spooler daemon acts as a sort of pipeline. It accepts print jobs coming in from various print clients, and then passes these jobs along to the appropriate printer. While the printer is busy, these jobs "spool," waiting for their chance to get printed.

When printing on the local printer, both of ends of this "pipeline" are described by the configuration file /etc/printcap (sometimes located at /etc/lprng/printcap). Each entry in the printcap (which is short for *printer capabilities*) describes one print spool:

```
$ more /etc/printcap
```

```
lp|Generic dot-matrix printer entry:\
        :lp=/dev/lp0:\
        :sd=/var/spool/lpd/lp:\
        :pl#66:\
        :pw#80:\
        :pc#150:\
        :mx#0:\
        :sh:
```

Note that the last line of the entry does **not** have a trailing backslash (\).

Your distribution may have other entries, and they may be more complex, but they should all have roughly this form. The name of this entry comes first, **lp**, followed by a longer description of this spool. The keyword/value pair **lp=/dev/lp0** specifies the Linux device were print jobs in this spool will be printed, and the **sd** keyword gives the directory where jobs will be held until they can be printed.

The rest of the keyword/value pairs provide details about what type of printer is hooked up to /dev/lp0. They are described in the *printcap* man page, and we'll cover some of them a little bit later.

Creating spool directories

If you create an entry, you'll need to make sure the spool directory exists and has the correct permissions. You want the printer daemon, which is usually run as user **lp**, to have access to the spool directory. You'll have to run these commands as root:

```
# mkdir -p /var/spool/lpd/lp
# chown lp /var/spool/lpd/lp
# chmod 700 /var/spool/lpd/lp
# checkpc -f
# /etc/init.d/lprng restart
```

LPRng includes a helpful tool for checking your printcap. It will even set up the spool directory for you, if you forget to do it manually:

```
# checkpc -f
```

Finally, restart the lpd. You'll need to do this any time you change the printcap, in order for your changes to take effect. You may need to use lpd instead of lprng:

```
# /etc/init.d/lprng restart
```

The older Berkeley printing system doesn't include a checkpc tool, so you'll have to try printing pages to your various printers yourself to make sure the printcap and spool directories are correct.

Using print spooler clients

The print spooler comes with several clients for communicating with the server daemon. The one you're likely to use most often is lpr, which simply sends a file to the server to be queued

up in a print spool and then printed. To try it out, first find or make a small sample text file. Then:

```
$ lpr sample.txt
```

If it worked, you shouldn't see any response on the screen, but your printer should start going, and soon you'll have a hard copy of your sample text. Don't worry if it doesn't come out looking quite right; we'll set up filters a bit later that should ensure that all sorts of file formats print correctly.

You can examine the list of print jobs in the print spool queue with the `lpq` command. The `-P` option specifies the name of the queue to display; if you leave it off, lpq will use the default spool, just like lpr did before:

```
$ lpq -Plp
Printer: lp@localhost 'Generic dot-matrix printer entry'
 Queue: 1 printable job
 Server: pid 1671 active
 Unspooler: pid 1672 active
 Rank   Owner/ID                  Class Job Files                 Size Time
active chouser@localhost+670        A   670 sample.txt               8 21:57:30
```

If you want to stop a job from printing, use the lprm command. You might want to do this if a job is taking to long, or if a user accidentally sends the same file more than once. Just copy the job id from the lpq listing above:

```
$ lprm chouser@localhost+670
Printer lp@localhost:
  checking perms 'chouser@localhost+670'
  dequeued 'chouser@localhost+670'
```

You can do many other operations on a print spool using the interactive tool lpc. See the *lpc* man page for details.

Printing to a remote LPD server

Even if you don't have a printer on your local machine, you can still use lpd to send a print job across the network to a printer that is attached to some other machine. On the client machine, you can add an entry to your /etc/printcap that looks like a local printer but actually routes print jobs to the server machine. This entry will look something like this:

```
farawaylp|Remote printer entry:\
        :rm=faraway:\
        :rp=lp:\
        :sd=/var/spool/lpd/farawaylp:\
        :mx#0:\
        :sh:
```

Here the name of the machine where we want to print is **faraway**, and the name of the printer on *that* machine is **lp**. The spool directory, /var/spool/lpd/farawaylp, is where print jobs will be held locally until they can be sent to the remote print spooler, where they may be spooled again before being sent to the printer. Again, you will need to create this spool

directory and set its permissions:

```
# mkdir -p /var/spool/lpd/farawaylp
# chown lp /var/spool/lpd/farawaylp
# chmod 700 /var/spool/lpd/farawaylp
# checkpc -f
# /etc/init.d/lprng restart
```

Locally, we have given this remote printer the name **farawaylp**, so we can send print jobs to it thusly:

```
$ lpr -Pfarawaylp sample.txt
```

Printing to a remote MS Windows or Samba server

Thanks to Samba, printing to a remote Microsoft Windows print server is only slightly more complicated. First, add the local printcap entry:

```
smb|Remote windows printer:\
        :if=/usr/bin/smbprint:\
        :lp=/dev/null:\
        :sd=/var/spool/lpd/smb:\
        :mx#0:
```

The new key here is **if**, the input filter. Pointing this to the smbprint script will cause the print job to be sent to a Windows server instead of the lp device. We still have to list a device (/dev/null in this case) which the print daemon uses for locking. But no print jobs will actually be sent there.

Don't forget to create the spool directory!

```
# mkdir -p /var/spool/lpd/smb
# chown lp /var/spool/lpd/smb
# chmod 700 /var/spool/lpd/smb
# checkpc -f
# /etc/init.d/lprng restart
```

In your favorite editor, create a .config file in the spool directory named above. In this case, /var/spool/lpd/smb/.config:

```
server="WindowsServerName"
service="PrinterName"
password=""
user=""
```

Adjust these values to point to the Windows machine and printer name that you want to use, and you're done:

```
$ lpr -Psmb sample.txt
```

The smbprint script should come with Samba, but it isn't included in all distributions. If you

can't find it on your system, you can get it from the *Samba HOWTO*.

Magicfilter

So far we've only tried printing text files, which isn't terribly exciting. Generally any one printer can only print one format of graphics file -- and yet there are dozens of different formats that we would like to print: PostScript, gif, jpeg, and many more. A program named *Magicfilter* acts as an input filter, much like smbprint does. It doesn't convert the file formats, rather it provides a framework for identifying the type of document you're trying to print, and runs it through an appropriate conversion tool: conversion tools must be installed separately. By far, the most important of these is *Ghostscript*, which can convert files from Postscript format to the native format of many printers.

Adjusting printcap to point to Magicfilter

Once these tools are installed, you simply need to adjust your printcap one more time. Add an **if** key to point to the Magicfilter that goes with your printer:

```
lp|The EPSON Stylus Color 777 sitting under my desk:\
        :if=/usr/share/magicfilter/StylusColor-777@720dpi-filter:\
        :gqfilter:\
        :lp=/dev/usb/lp0:\
        :sd=/var/spool/lpd/lp:\
        :pl#66:\
        :pw#80:\
        :pc#150:\
        :mx#0:\
        :sh:
```

There are filters for dozens and dozens of different printers and printer settings in /usr/share/magicfilter, so be sure you use the right one for your printer. Each of these filters is a text file, and usually the full name of the printer is at the top. This may help you if the file name of the filter isn't clear.

I also added a gqfilter flag to this printcap entry, which will cause the input filter to be used even when the print job is coming from a remote client. This only works with LPRng.

Since I set up the /var/spool/lpd/lp print spool directory earlier, I only need to check my printcap syntax and restart the server:

```
# checkpc -f
# /etc/init.d/lprng restart
```

Now you are able to print all sorts of documents, including Postscript files. In other words, choosing "Print" from a menu in your favorite web browser should now work.

Apsfilter as alternative to Magicfilter

Apsfilter provides many of the features of Magicfilter, but it also helps you set up your spool

directories, printcap entries, and so on. You will still need to make sure you have Ghostscript installed, but then you can follow the very complete instructions provided in the *Apsfilter handbook*.

Section 6. Resources and feedback

Resources

You can learn more about **configuring inetd and xinetd** from the article *Configuring inetd.conf securely* and from the *xinetd home page*. When adding your own service names and ports to /etc/services, remember to check first that they don't conflict with the *assigned port numbers*.

The *netfilter home page* is a good place to start learning more about **iptables and the Linux packet filter**. Until you're comfortable building your own ruleset, you might want to use an existing script for this. We recommend *gShield*.

Useful security tools include *Tripwire*, one of the most popular intrusion detection packages, and *Psionic Technologies*' *PortSentry*, which is a cross between intrusion prevention and detection. (The *LinuxWorld* article *How to stop crackers with PortSentry* offers advice on installation and configuration.) Finally, be sure to become familiar with Wietse Venema's *TCP Wrappers*, which allow monitoring of and control over connections to your system. You can view the *TCP wrappers README* online (it's also available in /usr/share/doc/tcp_wrappers-7.6). Authenticating users can be much easier with *Pluggable Authentication Modules* (also known as PAM).

Is your network wide open? Consider trying these two utilities for checking the vulnerability (or "openness") of your system: *netcat* is a simple UNIX utility that reads and writes data across network connections, using TCP or UDP protocol; *nmap* is a utility for network exploration or security auditing. Specifically, nmap scans ports to determine what's open.

Password checkers and other tools that will test how easy it is to guess your passwords (and those of your users) include *John the Ripper* from the *Openwall Project*, built for just this purpose. You may also want to try a comprehensive checker like *SAINT*.

These **security sites** should be among the most-visited by any systems administrator: *CERT* is a federally-funded center operated by Carnegie Mellon University. They study Internet security and vulnerabilities, publish security alerts, and research other security issues. *BugTraq*, hosted by Security Focus, is a full-disclosure moderated mailing list for the detailed discussion and announcement of computer security vulnerabilities. Even if you aren't particularly interested in this aspect of administration, a subscription to this list can be quite valuable, as simply scanning subject lines may alert you to vulnerabilities on your own systems that you might otherwise discover much later, or not at all.

Some **more security sites** we recommend highly for getting a better grip on the security of your Linux machines are the *Linux Security HOWTO*, *O'Reilly's Security Page*, and of course developerWorks' *Security zone* (although it has a greater emphasis on secure **programming** practices, it does also feature administrative security).

Printing and spooling resources you'll want to check out include the *LPRng print spooler home page* and the *Spooling Software overview* from the Printing HOWTO. Of course, the *Printing HOWTO* itself is a valuable resource, as is *LinuxPrinting.org*.

For help with **specific printers**, consult the *Serial HOWTO*. Also the *USB guide* offers valuable information on (you guessed it) USB printers.

Samba is a great help in heterogeneous networks. When setting up printing for this kind of environment, you will want to check out the *Samba home page* as well as the *Samba HOWTO*, with good printer sharing details.

The two **printer filters** we discussed were *Magicfilter* and *Apsfilter*. Remember that both need a conversion program (we recommend *Ghostscript*), as they do not do conversion themselves. If you are using the latter filter, you may also find the *Apsfilter handbook* to be quite useful.

In addition, we recommend the following general resources for learning more about Linux and preparing for LPI certification in particular:

You'll find a wealth of guides, HOWTOs, FAQs, and man pages at *http://www.linuxdoc.org*. Be sure to check out *Linux Gazette* and *LinuxFocus* as well.

The Linux Network Administrator's 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.

In the *Bash by example* article series on *developerWorks*, learn how to use `bash` programming constructs to write your own `bash` scripts. This series (particularly parts 1 and 2) are excellent additional preparation for the LPI exam:

- *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*

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

If you're not too familiar with the `vi` editor, you should check out Daniel's *tutorial on Vi*. This *developerWorks* 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`.

For more information on the Linux Professional Institute, visit the *LPI home page*.

Your feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact the lead author, 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.

LPI certification 102 exam prep, Part 4

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 "Secure shell and file sharing", the fourth of four tutorials designed to prepare you for the Linux Professional Institute's 102 exam. In this tutorial, we'll introduce you to the secure shell (ssh) and related tools and show you how to use and configure Network File System (NFS) version 3 servers and clients. By studying this series of tutorials (eight in all; this is part eight), you'll have the knowledge you need to become a Linux Systems Administrator and will be ready to attain an LPIC Level 1 certification (exams 101 and 102) from the Linux Professional Institute if you so choose.

The LPI logo is a trademark of the *Linux Professional Institute*.

Should I take this tutorial?

This tutorial is ideal for those who want to learn about or improve their basic Linux networking and file sharing skills. It's especially appropriate for those who will be setting up applications on Linux servers or desktops. For many, 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 important Linux system administration skills.

If you are new to Linux, we recommend that you first complete the previous tutorials in the LPI certification 101 and 102 exam prep series before continuing:

- *101 series, Part 1: Linux fundamentals*
- *101 series, Part 2: Basic administration*
- *101 series, Part 3: Intermediate administration*
- *101 series, Part 4: Advanced administration*
- *102 series, Part 1: Compiling sources and managing packages*
- *102 series, Part 2: Compiling and configuring the kernel*
- *102 series, Part 3: Networking*

About the author

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org*.

Residing in Albuquerque, New Mexico, Daniel Robbins 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 to 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.

Section 2. Secure shell

Interactive logins

Back in the old days, if you wanted to establish an interactive login session over the network, you used `telnet` or `rsh`. However, as networking became more popular, these tools became less and less appropriate, because they're horrendously insecure.

The data going between the telnet client and server isn't encrypted, and can thus be read by anyone snooping the network. Not only that, but *authentication* (the sending of your password to the server) is performed in plain text, making it a trivial matter for someone capturing your network data to get instant access to your password. In fact, using a network sniffer, it's possible for someone to reconstruct your entire telnet session, seeing everything on the screen that you saw!

Obviously, these tools that were designed with the assumption that the network was secure and unsniffable are inappropriate for today's distributed and public networks.

Secure shell

A better solution was needed, and that solution came in the form of a tool called secure shell, or `ssh`. The most popular modern incarnation of this tool is available in the openssh package, available for virtually every Linux distribution, not to mention many other systems.

What sets `ssh` apart from its insecure cousins is that it *encrypts* all communications between the client and the server using strong encryption. By doing this, it becomes difficult (impossible, even) to monitor the communications between the client and server. In this way, `ssh` provides its service as advertised -- it is a *secure* shell. In fact, `ssh` has excellent "all-round" security -- even authentication takes advantage of encryption and various key exchange strategies to ensure that the user's password cannot be easily grabbed by anyone monitoring data being transmitted over the network.

In this age of the popularization of the Internet, `ssh` is a valuable tool for enhancing network security when using Linux systems. Most security-savvy network admins discourage the use of -- or even don't allow the use of -- telnet and rsh on their systems at all because `ssh` is such a capable and secure replacement.

Using ssh

Generally, most distributions' openssh packages can be used without any manual configuration. After installing openssh, you'll have a couple of binaries. One is, of course, `ssh` -- the secure shell client that can be used to connect to any system running `sshd`, the secure shell server. To use `ssh`, you typically start a session by typing something like:

```
$ ssh drobbins@otherbox
```

Above, I instruct `ssh` to login as the "drobbins" user account on remotebox. As with telnet, you'll be prompted for a password; after entering it, you'll be presented with a new login session on the remote system.

Starting sshd

If you want to allow `ssh` connections to your machine, you'll need to start the `sshd` server. To start the `sshd` server, you would typically use the rc-script that came with the openssh package, typing something like:

```
# /etc/init.d/sshd start
```

or

```
# /etc/rc.d/init.d/sshd start
```

If necessary, you can adjust configuration options for `sshd` by modifying the /etc/ssh/sshd_config file. For more information on the various options available, type `man sshd`.

Secure copy

The openssh package also comes with a handy tool called `scp`, which stands for "secure copy". You can use this command to securely copy files to and from various systems on the network. For example, if I wanted to copy ~/foo.txt to my home directory on remotebox, I could type:

```
$ scp ~/foo.txt drobbins@remotebox:
```

After being prompted for my password on remotebox, the copy will be performed. Or, if I wanted to copy a file called bar.txt in remotebox's /tmp directory to my current working directory on my local system, I could type:

```
$ scp drobbins@remotebox:/tmp/bar.txt .
```

Secure shell authentication options

Openssh also has a number of other authentication methods. Used properly, they can allow you to authenticate with remote systems without having to type in a password or passphrase for every connection. To learn more about how to do this, read the *developerWorks* openssh key management articles (listed in Resources, the last section of this tutorial).

Section 3. NFS

Introducing NFS

The Network File System (NFS) is a technology that allows the transparent sharing of files between UNIX and Linux systems connected via a Local Area Network, or LAN. NFS has been around for a long time; it's well known and used extensively used in the Linux and UNIX worlds. In particular, NFS is often used to share home directories among many machines on the network, providing a consistent environment for a user when he or she logs in to a machine (*any* machine) on the LAN. Thanks to NFS, it's possible to mount remote filesystem trees and have them fully integrated into a system's local filesystem. NFS' transparency and maturity is what makes it such a useful and popular choice for network file sharing under Linux.

NFS basics

To share files using NFS, you first need to set up an NFS server. This NFS server can then "export" filesystems. When a filesystem is exported, it means that it is made available to be accessed by other systems on the LAN. Then, any authorized system that is also set up as an NFS client can mount this exported filesystem using the standard "mount" command. After the mount completes, the remote filesystem is "grafted in" in the same way that a locally-mounted filesystem (like /mnt/cdrom) would be after it is mounted. The fact that all of the file data is being read from the NFS server rather than from a disk is simply not an issue to any standard Linux application. Everything simply works.

Attributes of NFS

Shared NFS filesystems have a number of interesting attributes. The first "nifty attribute" is a result of NFS' stateless design. Because client access to the NFS server is stateless in nature, it's possible for the NFS server to reboot without causing client applications to crash or fail. All access to remote NFS files will simply "pause" until the server comes back online. Also, because of NFS' stateless design, NFS servers can handle large numbers of clients without any additional overhead besides that of transferring the actual file data over the network. In other words, NFS performance is dependant on the amount of NFS data being transferred over the network, rather than the number of machines that happen to be requesting said data.

NFS version 3 under Linux

When setting up NFS, it's strongly recommended that you use NFS version 3 rather than version 2. Version 2 has some significant problems with file locking and generally has a bad reputation for breaking certain applications. On the other hand, NFS version 3 is very nice and robust and does its job well. Now that Linux 2.2.18+ supports NFS 3 clients and servers, there's no reason at all to consider using NFS 2 anymore.

Securing NFS

It's important to mention that NFS version 2 and 3 have some very clear security limitations. They were designed to be used in a specific environment -- a secure, trusted LAN. In particular, NFS 2 and 3 were designed to be used on a LAN where "root" access to the machine is only allowed by administrators. Due to the design of NFS 2 and NFS 3, if a malicious user has "root" access to a machine on your LAN, he or she will be capable of bypassing NFS security and very likely be able to access or even modify files on the NFS server that he or she wouldn't normally be able to otherwise. For this reason, NFS should not be deployed casually. If you're going to use NFS on your LAN, great -- but set up a firewall first. Make sure that people outside your LAN won't be able to access your NFS server. Then, make sure that your internal LAN is relatively secure, and that you are fully aware of all the hosts participating in your LAN. Once your LAN's security has been thoroughly reviewed and (if necessary) improved, you're ready to safely use NFS (see Part 7 of this tutorial series for more on this).

Section 4. Setting up NFS

Setting up NFS under Linux

The first step in using NFS 3 is to set up an NFS 3 server. Choose the system that will be serving files to the rest of your LAN. On this machine, we'll need to enable NFS server support in the kernel. You should use a 2.2.18+ kernel (2.4+ recommended) to take advantage of NFS 3, which is much more stable than NFS 2. If you're compiling your own custom kernel, enter your /usr/src/linux directory and run `make menuconfig`. Then, select the "File systems section," then the "Network File Systems" section, and ensure that the following options are enabled:

```
<*> NFS file system support
[*]   Provide NFSv3 client support
<*> NFS server support
[*]   Provide NFSv3 server support
```

Getting ready for /etc/exports

Next, compile and install your new kernel and reboot. Your system will now have NFS 3 server and client support built-in.

Now that our NFS server has support for NFS in the kernel, it's time to set up an /etc/exports file. The /etc/exports file will describe the local filesystems that will be made available for export, as well as which hosts will be able to access these filesystems, and whether they will be exported as read/write or read-only. It will also allow us to specify other options that control NFS behavior.

But before we look at the format of the /etc/exports file, a big fat implementation warning is in order! The NFS implementation in the Linux kernel only allows the export of one local directory per filesystem. This means that if both /usr and /home are on the same ext3 filesystem (using /dev/hda6, for example), then you can't have both /usr and /home export lines in /etc/exports. If you try to add these lines, you'll see errors like this when your /etc/exports file gets reread (which will happen if you type `exportfs -ra` after your NFS server is up and running):

```
sidekick:/home: Invalid argument
```

Working around export restrictions

Here's how to work around this problem. If /home and /usr are on the same underlying local filesystem, you can't export them both. So just export /. NFS clients will then be able to mount /home and /usr via NFS just fine, but your NFS server's /etc/exports file will now be "legal," containing only one export line per underlying local filesystem. Now that you understand this implementation of Linux NFS, we're ready to take a look at the format of /etc/exports.

The /etc/exports file

Probably the best way to understand the format of /etc/exports is to look at a quick example. Here's a simple /etc/exports file that I use on my NFS server:

```
# /etc/exports: NFS file systems being exported.  See exports(5).
/ 192.168.1.9(rw,no_root_squash)
/mnt/backup 192.168.1.9(rw,no_root_squash)
```

As you can see, the first line in my /etc/exports file is a comment. On the second line, I select my root ("/") filesystem for export. Note that while this exports everything under "/", it will not export any other local filesystem. For example, if my NFS server has a CD-ROM mounted at /mnt/cdrom, the contents of the CDROM will not be available unless they are exported explicitly in /etc/exports. Now, notice the third line in my /etc/exports file. On this line, I export /mnt/backup; as you might guess, /mnt/backup is on a separate filesystem from /, and it contains a backup of my system. Each line also has a "192.168.1.9(rw,no_root_squash)" on it. This information tells nfsd to only make these exports available to the NFS client with the IP address of 192.168.1.9. It also tells nfsd to make these filesystems writeable as well as readable by NFS client systems, and instructs the NFS server to allow the remote NFS client to allow a superuser account to have true "root" access to the filesystems.

Another /etc/exports file

Here's an /etc/exports that will export the same filesystems as the one in the previous panel, except that it will make my exports available to all machines on my LAN -- 192.168.1.1 through 192.168.1.254:

```
# /etc/exports: NFS file systems being exported.  See exports(5).
/ 192.168.1.1/24(rw,no_root_squash)
/mnt/backup 192.168.1.1/24(rw,no_root_squash)
```

In this sample /etc/exports file, I use a host mask of /24 to mask out the last eight bits in the IP address I specify. It's also very important that there is no space between the IP address specification and the "(", or NFS will interpret your information incorrectly. And, as you might guess, there are other options that one can specify besides "rw" and "no_root_squash"; type "man exports" for a complete list.

Starting the NFS 3 server

Once /etc/exports is configured, you're ready to start your NFS server. Most distributions will have an "nfs" initialization script that can be used to start NFS -- type `/etc/init.d/nfs start` or `/etc/rc.d/init.d/nfs start` to use it. Once NFS is started, typing `rpcinfo` should display output that looks something like this:

```
# rpcinfo -p
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  32802  status
    100024    1   tcp  46049  status
```

```
    100011    1   udp    998  rquotad
    100011    2   udp    998  rquotad
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    100003    2   tcp   2049  nfs
    100003    3   tcp   2049  nfs
    100021    1   udp  32804  nlockmgr
    100021    3   udp  32804  nlockmgr
    100021    4   udp  32804  nlockmgr
    100021    1   tcp  48026  nlockmgr
    100021    3   tcp  48026  nlockmgr
    100021    4   tcp  48026  nlockmgr
    100005    1   udp  32805  mountd
    100005    1   tcp  39293  mountd
    100005    2   udp  32805  mountd
    100005    2   tcp  39293  mountd
    100005    3   udp  32805  mountd
    100005    3   tcp  39293  mountd
```

Changing export options

If you ever change your /etc/exports file while your NFS daemons are running, simply type `exportfs -ra` to apply your changes. Now that your NFS server is up and running, you're ready to configure NFS clients so that they can mount your exported filesystems:

Configuring NFS clients

Kernel configuration for NFS 3 clients is similar to that of the NFS server, except that you only need to ensure that the following options are enabled:

```
<*> NFS file system support
[*]   Provide NFSv3 client support
```

Starting NFS client services

To start the appropriate NFS client daemons, you can typically use a system initialization script called "nfslock" or "nfsmount". Typically, this script will start rpc.statd, which is all the NFS 3 client needs -- rpc.statd allows file locking to work properly. Once all your client services are set up, running `rpcinfo` on the local machine will display output that looks like this:

```
# rpcinfo
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  32768  status
    100024    1   tcp  32768  status
```

You can also perform this check from a remote system by typing `rpcinfo -p myhost`, as follows:

```
# rpcinfo -p sidekick
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100024    1   udp  32768  status
    100024    1   tcp  32768  status
```

Mounting exported NFS filesystems

Once both client and server are set up correctly (and assuming that the NFS server is configured to allow connections from the client), you can go ahead and mount an exported NFS filesystem on the client. In this example, `inventor` is the NFS server and `sidekick` (IP address 192.168.1.9) is the NFS client. Inventor's /etc/exports file contains a line that looks like this, allowing connections from any machine on the 192.168.1 network:

```
/ 192.168.1.1/24(rw,no_root_squash)
```

Now, logged into `sidekick` as root, we can type:

```
# mount inventor:/ /mnt/nfs
```

Inventor's root filesystem will now be mounted on sidekick at /mnt/nfs; you should now be able to type `cd /mnt/nfs` and look around inside and see inventor's files. Again, note that if inventor's /home tree is on another filesystem, then /mnt/nfs/home will not contain anything -- another `mount` (as well as another entry in inventor's /etc/exports file) will be required to access that data.

Mounting directories *inside* exports

Note that inventor's `/ 192.168.1.1/24(rw,no_root_squash)` line will also allow us to mount directories *inside* /. For example, if inventor's /usr is on the same physical filesystem as /, and you are only interested in mounting inventor's /usr on sidekick, you could type:

```
# mount inventor:/usr /mnt/usr
```

Inventor's /usr tree will now be NFS mounted to the pre-existing /mnt/usr directory. It's important to again note that inventor's /etc/exports file didn't need to explicitly export /usr; it was included "for free" in our "/" export line.

Section 5. Resources and feedback

Resources

This wraps up not only this tutorial, but also our LPI 102 exam series. We hope you've enjoyed the ride! While the tutorial is over, learning never ends. You'll find useful instruction in the following resources, particularly if you plan to take the LPI 102 exam:

The best thing you can do to improve your NFS skills is to try setting up your own NFS 3 server and client(s) -- the experience will be invaluable. The second-best thing you can do is to read *the Linux NFS HOWTO*, which is quite a good HOWTO.

Learn more about what ssh is capable of in the *developerWorks* series on ssh: *Part 1*, *Part 2*, and *Part 3*. Also be sure to visit the home of openssh at *http://www.openssh.com*, which is an excellent place to continue your study of this important tool.

Samba is another important networked file-sharing technology. For more information about Samba, read the *developerWorks* Samba articles: the *first "Key concepts" article*, *Samba installation* article, and *Samba configuration* article.

Once you're up to speed on Samba, spend some time studying the *Linux DNS HOWTO*. The LPI 102 exam is also going to expect that you have some familiarity with Sendmail. Red Hat has a good *Red Hat Sendmail HOWTO* that will help to get you up to speed.

Be sure to also check out the following general resources:

http://www.linuxdoc.org has an excellent collection of guides, HOWTOs, FAQs, and man-pages. While there, be sure to check out *Linux Gazette* and *LinuxFocus*.

The Linux Network 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.

In the *Bash by example* article series on *developerWorks*, learn how to use `bash` programming constructs to write your own `bash` scripts. This series (particularly parts 1 and 2) are excellent additional preparation for the LPI exam:

- *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*

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

If you're not too familiar with the `vi` editor, you should check out Daniel's *tutorial on Vi*. This *developerWorks* 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`.

For more information on the Linux Professional Institute, visit the *LPI home page*.

Your feedback

We look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact the lead author, 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.