99 Pages to™

Preface

This book is a collection from various online and offline sources. It is assembled from publicly available contents as a free service to the community, and organized in a way that simplifies progressive understanding of Linux fundamentals while keeping it structured for use as a quick reference for your toolset.

The contents are independent of distributions and should apply to most Linux systems; however, it is always useful to check man pages from your specific vendor.

The next 99 pages will take you from absolute beginner to advanced user in 4 easy sections. The detailed table of contents is a quick way to locate information so you can use this as a reference.

Click here to skip the table of contents and get right to the book.

Now let's dig right into Linux…

Table of Contents

To begin, let's 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 inodes and hard and symbolic links. By the end of this section, you'll have a solid grounding in Linux fundamentals and will even be ready to begin learning some basic Linux system administration tasks.

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 systems host name, the name of the current working directory, or both. But regardless of what your prompt looks 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
```

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 log out 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 file system. 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 file system 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.

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

Understanding "."

Before we finish our coverage of cd, there are a few more things I need to mention. 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:

```
$ ./someprogram
```

In the above example, the someprogram 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 someprogram command. If the file lives in our home directory, we can type:

```
$ ./ someprogram /home/joe/somefile.txt
```

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

```
$ ./someprogram ~/somefile.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 joesfile.txt in Joe's home directory, we could type:

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

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
bin  games  include  keystoneclient  lib  local  sbin  share  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  games  include  keystoneclient  lib  local  sbin  share  src
```

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.

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

```
$ ls -l /usr
total 72
drwxr-xr-x   2 root root 32768 Aug 22 05:41 bin
drwxr-xr-x   2 root root  4096 Apr 19  2012 games
drwxr-xr-x  32 root root  4096 Aug 22 05:41 include
drwxr-xr-x   2 root root  4096 Aug 22 04:32 keystoneclient
drwxr-xr-x  66 root root 12288 Aug 22 05:41 lib
drwxr-xr-x  10 root root  4096 May  1  2012 local
drwxr-xr-x   2 root root  4096 Aug 22 05:41 sbin
drwxr-xr-x 123 root root  4096 Aug 22 05:42 share
drwxr-xr-x   5 root root  4096 Aug 22 04:49 src
```

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 file system 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/games ../home
drwxr-xr-x  3 root root  4096 Aug 22 04:13 ../home
drwxr-xr-x 11 root root  4096 Aug 22 04:32 /usr
drwxr-xr-x  2 root root 32768 Aug 22 05:41 /usr/bin
drwxr-xr-x  2 root root  4096 Apr 19  2012 /usr/games
```

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 files and 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 file system objects in the listing:

```
$ ls -i /usr
409602 bin    409604 include         409605 lib    409607 sbin   409609 src
409603 games  582841 keystoneclient  409606 local  409608 share
```

Understanding inodes

Every object on a file system is assigned a unique index, called an inode number. This might seem trivial, but understanding inodes is essential to understanding many file system 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
409606 /usr/local
```

The /usr/local directory has an inode number of 409606. Now, let's take a look at the inode number of /usr/local/bin/..:

```
$ ls -id /usr/local/bin/..
409606 /usr/local/bin/..
```

As you can see, /usr/local/bin/.. has the same inode number as /usr/local! Here's how we can 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 409606 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 409606. Although inode 409606only exists in one place on disk, multiple things link to it. Inode 409606 is the actual entry on disk.

In fact, we can see the total number of times that inode 409606 is referenced by using the

`ls -dl`
command:

```
$ ls -dl /usr/local
drwxr-xr-x 10 root root 4096 May  1  2012 /usr/local
```

If we take a look at the second column from the left, we see that the directory /usr/local (inode 409606) is referenced ten 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 one two three
```

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 **bea/uti/ful**, you'd need to issue three separate mkdir commands:

```
$ mkdir bea/uti/ful
mkdir: cannot create directory `bea/uti/ful': No such file or directory
$ mkdir bea
$ mkdir bea/uti
$ mkdir bea/uti/ful
```

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

```
$ mkdir -p bea/uti/ful
```

To learn more about the mkdir command, type man mkdir to read the manual page. This will work for nearly all commands covered here (for example, 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

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
```

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 file system as the source file.

While we're talking about mv, let's look at another way to use this command. mv, in addition to allowing us to rename files, also allows us to move one or more files to another location in the directory hierarchy. For example, to move **/var/tmp/myfile.txt** to **/home/joe**, you could type:

```
$ mv /var/tmp/myfile.txt /home/joe
```

After typing this command, myfile.txt will be moved to **/home/joe/myfile.txt**. And if **/home/joe** is on a different file system than /var/tmp, the mv command will handle the copying of myfile.txt to the new file system and erasing it from the old file system. As you might guess, when myfile.txt is moved between file systems, the myfile.txt at the new location will have a new inode number. This is because every file system has its own independent set of inode numbers.

We can also use the mv command to move multiple files to a single destination directory. For example, to move myfile1.txt and myarticle3.txt to /home/joe, I could type:

```
$ mv /var/tmp/myfile1.txt /var/tmp/myarticle3.txt /home/joe
```

Creating Links and Removing Files

Hard links

We've mentioned the term "link" when referring to the relationship between directory entries (the "names" we type) and inodes (the index numbers on the underlying file system that we can usually ignore.) 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 file system until all the hard links disappear. When the last hard link disappears and no program is holding the file open, Linux will delete the file automatically. New hard links can be created using the ln command:

```
$ cd /tmp
$ touch firstlink
$ ln firstlink secondlink
$ ls -i firstlink secondlink
  868361 firstlink  868361 secondlink
```

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 (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 file systems; which would be the case if the file systems are on separate disk partitions. This means that you can't create a link from /usr/bin/bash to /bin/bash if your / and /usr directories exist on separate disk partitions.

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.

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

```
$ ln -s secondlink thirdlink
$ ls -l firstlink secondlink thirdlink
-rw-r--r-- 2 root root  0 Sep 19 05:36 firstlink
-rw-r--r-- 2 root root  0 Sep 19 05:36 secondlink
lrwxrwxrwx 1 root root 10 Sep 19 05:38 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 preceded by a cute little ->.

Symlinks in-depth

Symbolic links are generally more flexible than hard links. You can create a symbolic link to any type of file system 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 physical file system; that is, a different disk partition. However, this fact can also make symbolic links tricky to understand.

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 15 Sep 19 05:39 bin1 -> /usr/local/bin/
```

Or alternatively:

```
$ ln -s ../usr/local/bin bin2
$ ls -l bin2
lrwxrwxrwx 1 root root 16 Sep 19 05:40 bin2 -> ../usr/local/bin
```

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 Sep 19 05:40 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.

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/dig
-rwxr-xr-x 1 root root 124960 Jul 27 03:17 /usr/bin/dig
```

As the root user, you may want to create an alternate name for "dig", such as "dg". 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 dig as follows:

```
# cd /usr/bin
# ln -s /usr/bin/dig dg
# ls -l dig dg
lrwxrwxrwx 1 root root     12 Sep 19 05:45 dg -> /usr/bin/dig
-rwxr-xr-x 1 root root 124960 Jul 27 03:17 dig
```

In this example, we created a symbolic link called dg that points to the file /usr/bin/dig.

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

```
# mv /usr/bin/dig /usr/bin/dg /usr/local/bin
# ls -l /usr/local/bin/dig
-rwxr-xr-x    1 root     root        10150 Dec 12 20:09 /usr/local/bin/dig
# ls -l /usr/local/bin/dg
lrwxrwxrwx    1 root     root           17 Mar 27 17:44 dg -> /usr/bin/dig
```

Because we used an absolute path in our symbolic link, our dg symlink is still pointing to /usr/bin/dig, which no longer exists since we moved /usr/bin/dig to /usr/local/bin.

That means that dg is now a 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. The following example would have worked even after both files were moved:

```
# cd /usr/bin
# ln -s dig dg
# ls -l dg
lrwxrwxrwx    1 root     root            8 Jan  5 12:40 dg -> dig
# mv dig dg /usr/local/bin
# ls -l /usr/local/bin/dig
-rwxr-xr-x    1 root     root        10150 Dec 12 20:09 /usr/local/bin/dig
# ls -l /usr/local/bin/dg
lrwxrwxrwx    1 root     root           17 Mar 27 17:44 dg -> dig
```

Now, we can run the dig program by typing /usr/local/bin/dg. /usr/local/bin/dg points to the program keychain in the same directory as dg.

rm

Now that we know how to use cp, mv, and ln, it's time to learn how to remove objects from the file system. 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
```

Note that under Linux, once a file is rm'ed, it's typically gone forever. For this reason, many junior system administrators will use the -i option when removing files. The -i option tells rm to remove all files in interactive mode -- that is, prompt before removing any file. For example:

```
$ rm -i file1 file2
rm: remove regular empty file `file1'? y
rm: remove regular empty file `file2'? y
```

In the above example, the rm command prompted whether or not the specified files should *really* be deleted. In order for them to be deleted, I had to type "y" and Enter twice. If I had typed "n", the file would not have been removed. Or, if I had done something really wrong, I could have typed Control-C to abort the rm -i command entirely -- all before it is able to do any potential damage to my system.

If you are still getting used to the rm command, it can be useful to add the following line to your ~/.bashrc file using your favorite text editor, and then log out and log back in. Then, any time you type rm, the bash shell will convert it automatically to an rm -i command. That way, rm will always work in interactive mode:

```
alias rm="rm -i"
```

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
```

This method is commonly referred to as "directory removal for suckers." All real power users and administrators worth their salt use the much more convenient rm -rf command, covered next.

The best way to remove a directory is to 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 :)

Using Wild cards

Introducing Wild cards

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 file system 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
```

To solve this problem, you can take advantage of Linux' built-in wild card 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 as well as any file named file, you could type:

```
$ rm file*
```

The * wildcard matches any character or sequence of characters, or even "no character." Of course, glob wildcards can be used for more than simply removing files, as we'll see in the next panel.

Understanding non-matches

If you wanted to list all the file system objects in /etc beginning with g as well as any file called g, you could type:

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

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

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

Here's what happened. Normally, when we specify a pattern, that pattern matches one or more files on the underlying file system, 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, wild cards and all, as-is*. So, then ls can't find the file /usr/bin/asdf*jkl and it gives us an error. The operative rule here is that *glob patterns are expanded only if they match objects in the file system*. Otherwise they remain as is and are passed literally to the program you're calling.

Wild card syntax: "*" and "?"

Now that we've seen how globbing works, we should look at wild card syntax. You can use special characters for wild card expansion:

"*" will match zero or more characters. It means "anything can go here, including nothing". Examples:

- /etc/g* matches all files in /etc that begin with g, or a file called g.
- /tmp/my*1 matches all files in /tmp that begin with my and end with 1, including the file my1.

"?" 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

Wild card syntax: "[]"

This wild card is like a "?", but it allows more specificity. To use this wild card, 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 wild card 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 wild cards 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.

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]. Example:

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

Wild card caveats

Here are some caveats to watch out for when using wild cards. Since bash treats wild card-related characters (?, [,], and *) 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 wild card expansion on them:

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

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 wild cards:

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

Both approaches (single quotes and backslash escaping) have the same effect. 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 \).

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 wild card expansion, type man 7 glob. For more information on quoting in bash, type man 8 glob and read the section titled QUOTING.

Basic Administration

In this section, 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 (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. Next, we'll give you a whirlwind introduction to shell pipelines, redirection, and text processing commands. Finally, we'll introduce you to Linux kernel modules.

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 to perform search-and-replace operations on text streams.

Glob comparison

As we take a look at regular expressions, you may find that regular expression syntax looks similar to the filename "globbing" syntax that we looked at in previous section. 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 take a 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
root:x:0:0:root:/root:/bin/bash
mystack:x:1000:1000:,,,:/home/mystack:/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 to be 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.sda /etc/fstab
/dev/sda3       /               reiserfs        noatime,ro 1 1
/dev/sda1       /boot           reiserfs        noauto,noatime,notail 1 2
/dev/sda2       swap            swap            sw 0 0
#/dev/sda4      /mnt/extra      reiserfs        noatime,rw 1 1
```

In this example, the literal text dev.sda didn't appear on any of the lines in /etc/fstab. However, grep wasn't scanning them for the literal dev.sda string, but for the dev.sda 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.sda[12] /etc/fstab
/dev/sda1       /boot           reiserfs        noauto,noatime,notail 1 2
/dev/sda2       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.sda[^12] /etc/fstab
/dev/sda3       /               reiserfs        noatime,ro 1 1
#/dev/sda4      /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.sda by typing:

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

Alternately, we could also type:

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

Neither regular expression is 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. Note that this means that the * has a different meaning in a regex than it does with globs. Here are some examples, and play close attention to instances where these regex matches differ from globs:

- ab*c matches abbbbc but not abqc (if a glob, it would match both strings -- can you figure out why?)
- ab*c matches abc but not abbqbbc (again, if a glob, it would match both strings)
- ab*c matches ac but not cba (if a glob, ac would not be matched, nor would cba)
- b[cq]*e matches bqe and be (if a glob, it would match bqe but not be)
- b[cq]*e matches bccqqe but not bccc (if a glob, it would match the first but not the second as well)
- b[cq]*e matches bqqcce but not cqe (if a glob, it would match the first but not the second as well)
- b[cq]*e doesn't match bbbeee (this would not be the case with a glob)
- .* will match any string. (if a glob, it would match any string starting with .)
- foo.* will match any string that begins with foo (if a glob, it would match any string starting with the four literal characters foo..)

Now, for a quick brain-twisting review: the line ac matches the regex ab*c because the asterisk also allows the preceding expression (b) to appear zero times. Again, it's critical to 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 regexes

^ 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.

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 -- so that stuff is in generally the same place across Linux distributions. The FHS specifies the following directory tree (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 /opt	/etc /boot
variable	/var/mail /var/spool/news	/var/run /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**.

More information on Filesystem Hierarchy Standard can be found at http://www.pathname.com/fhs/.

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. This introduction will help you 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/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
```

Given this value of PATH (yours may differ,) bash would first check **/usr/local/bin**, then **/usr/local/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 to it on the command line:

```
$ PATH=$PATH:~/bin
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/home
/joe/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/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

To make your PATH changes available to any future processes you start from this shell, export your changes using the export command:

```
$ export PATH
```

All about "which"

You can check to see if there's a given program in your PATH by using which:

```
$ 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 handy 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/ifupdown/README
/usr/share/doc/libavahi-common3/README
/usr/share/doc/libgirepository-1.0-1/README
/usr/share/doc/libconfig-general-perl/README
/usr/share/doc/libtasn1-3/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/ifupdown/README
/usr/share/doc/libavahi-common3/README
/usr/share/doc/libgirepository-1.0-1/README
/usr/share/doc/libconfig-general-perl/README
/usr/share/doc/libtasn1-3/README
/usr/share/doc/fakeroot/README
/usr/share/doc/python-pam/README
/usr/share/doc/ltrace/README
/usr/share/doc/libjs-underscore/README
/usr/share/doc/python-m2crypto/README
[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:

```
$ find /etc -iregex '.*xt.*'
/etc/ssl/certs/AddTrust_External_Root.pem
/etc/update-motd.d/10-help-text
/etc/apache2/mods-available/ext_filter.load
/etc/apparmor.d/abstractions/ubuntu-xterm
/etc/apparmor.d/abstractions/ubuntu-browsers.d/text-editors
/etc/alternatives/text.plymouth
```

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 as the regex 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 contains the string vim:

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

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 2003
```

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 100 bytes:

```
$ find /usr/bin -type f -size -100c
/usr/bin/paster
/usr/bin/rgrep
/usr/bin/pydoc2.7
/usr/bin/2to3-2.7
/usr/bin/paster2.7
/usr/bin/cheetah
/usr/bin/cheetah-compile
/usr/bin/migrate
```

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 it replaces any occurrences of {} with the filename. This is best understood with an example:

```
$ find /usr/bin -type f -size -100c -exec ls -l '{}' ';'
-rwxr-xr-x 1 root root 66 Nov  9  2011 /usr/bin/paster
-rwxr-xr-x 1 root root 30 Jul  6  2011 /usr/bin/rgrep
-rwxr-xr-x 1 root root 79 Apr 10 06:46 /usr/bin/pydoc2.7
-rwxr-xr-x 1 root root 96 Apr 10 06:46 /usr/bin/2to3-2.7
-rwxr-xr-x 1 root root 69 Nov  9  2011 /usr/bin/paster2.7
-rwxr-xr-x 1 root root 73 Dec 18  2011 /usr/bin/cheetah
-rwxr-xr-x 1 root root 89 Dec 18  2011 /usr/bin/cheetah-compile
-rwxr-xr-x 1 root root 90 Feb 19  2008 /usr/bin/migrate
```

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, since 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 generated by updatedb (which we'll cover in the next panel.)

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

```
$ locate bin/ls
/bin/ls
/bin/lsblk
/bin/lsmod
/sbin/lsmod
/usr/bin/lsattr
/usr/bin/lsb_release
/usr/bin/lscpu
/usr/bin/lshw
/usr/bin/lsinitramfs
/usr/bin/lsof
/usr/bin/lspci
/usr/bin/lspgpot
/usr/bin/lsusb
/usr/lib/klibc/bin/ls
```

Using updatedb

Most Linux systems have a "cron job" to update the database periodically. If your locate returned an error such as the following, then you will need to run updatedb as root 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

On 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 would otherwise be unable 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.

Process Control

Staring xeyes

You may need to install xeyes on your system first. Consult your distro's documentation for instructions on installing

To learn about process control, we first need to start a process. Make sure that you have X running and execute the following command:

```
$ 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 in your terminal.

Stopping a process

To get a prompt back, you could type Control-C (often written as Ctrl-C or ^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
Control-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
(test it out, then stop the process again)
Control-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 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; for example, 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 &
[1] 15632
$ 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 ?        Ss     0:10 /sbin/init
    2 ?        S      0:00 [kthreadd]
    3 ?        S     23:43 [ksoftirqd/0]
    4 ?        S      0:00 [kworker/0:0]
    5 ?        S     23:19 [kworker/u:0]
    6 ?        S      3:27 [migration/0]
    7 ?        S      0:14 [watchdog/0]
    8 ?        S      3:24 [migration/1]
    9 ?        S      0:00 [kworker/1:0]
   10 ?        S     24:42 [ksoftirqd/1]
   11 ?        S      0:13 [watchdog/1]
   12 ?        S<     0:00 [cpuset]
    .
    .
```

I've only listed the first few because it is usually a very long list. This gives you a snapshot of what the whole machine is doing, but is a lot of information to sift through. If you were to leave off 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
 7215 ?        S      0:03 dnsmasq --no-hosts --no-resolv --strict-order --
bind-
    2 ?        S      0:00 [kthreadd]
    3 ?        S     23:43  \_ [ksoftirqd/0]
    4 ?        S      0:00  \_ [kworker/0:0]
    5 ?        S     23:19  \_ [kworker/u:0]
    6 ?        S      3:27  \_ [migration/0]
    7 ?        S      0:14  \_ [watchdog/0]
    8 ?        S      3:24  \_ [migration/1]
    9 ?        S      0:00  \_ [kworker/1:0]
    .
    .
```

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
root       919  0.0  0.0 15780   764 tty4     Ss+  Aug25   0:00 /sbin/getty -8
root       925  0.0  0.0 15784   764 tty5     Ss+  Aug25   0:00 /sbin/getty -8
root       952  0.0  0.0 15784   764 tty2     Ss+  Aug25   0:00 /sbin/getty -8
root       953  0.0  0.0 15784   764 tty3     Ss+  Aug25   0:00 /sbin/getty -8
root       960  0.0  0.0 15784   764 tty6     Ss+  Aug25   0:00 /sbin/getty -8
root      2465  0.0  0.0 12752   832 hvc0     Ss+  Aug25   0:00 /sbin/getty -L
root      2468  0.0  0.0 15784   924 tty1     Ss+  Aug25   0:00 /sbin/getty -8
root      5532  0.0  0.0 18160  1260 pts/1    R+   17:31   0:00 ps au
root     25217  0.0  0.1 23108  4420 pts/1    Ss   05:07   0:00 -bash
```

```
$ ps al
F   UID   PID  PPID PRI NI    VSZ   RSS WCHAN  STAT TTY      TIME COMMAND
4     0   919     1  20  0  15780   764 n_tty_ Ss+  tty4     0:00 /sbin/getty
4     0   925     1  20  0  15784   764 n_tty_ Ss+  tty5     0:00 /sbin/getty
4     0   952     1  20  0  15784   764 n_tty_ Ss+  tty2     0:00 /sbin/getty
4     0   953     1  20  0  15784   764 n_tty_ Ss+  tty3     0:00 /sbin/getty
4     0   960     1  20  0  15784   764 n_tty_ Ss+  tty6     0:00 /sbin/getty
4     0  2465     1  20  0  12752   832 n_tty_ Ss+  hvc0     0:00 /sbin/getty
4     0  2468     1  20  0  15784   924 n_tty_ Ss+  tty1     0:00 /sbin/getty
4     0  8580 25217  20  0   9728  1028 -      R+   pts/1    0:00 ps al
4     0 25217 24872  20  0  23108  4420 wait   Ss   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
```

```
top - 17:39:00 up 25 days, 11:00,  1 user,  load average: 0.57, 0.79, 0.87
Tasks: 121 total,   2 running, 118 sleeping,   0 stopped,   1 zombie
Cpu(s): 14.6%us, 12.6%sy,  0.0%ni, 72.4%id,  0.0%wa,  0.0%hi,  0.0%si,
0.3%st
Mem:   4099408k total,  3949620k used,   149788k free,   248440k buffers
Swap:  1999996k total,    23024k used,  1976972k free,  2165728k cached

  PID USER       PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 1046 quantum    20   0  102m  28m 4240 S    4  0.7   1858:01 python
 2045 rabbitmq   20   0 1138m 111m 1920 S    1  2.8 533:28.01 beam.smp
17536 root       20   0 37936 7348 3064 R    1  0.2   0:00.04 python
 1040 cinder     20   0 91412  19m 3756 S    1  0.5 313:47.47 cinder-volume
 1037 nova       20   0 1440m 178m 5376 S    1  4.5  88:37.65 nova-compute
 1024 quantum    20   0  187m  41m 4508 S    0  1.0 282:44.35 python
 1062 nova       20   0  228m  64m 4612 S    0  1.6 179:50.30 nova-conductor
 1071 quantum    20   0  101m  29m 4240 S    0  0.7  41:24.57 python
 1617 root       10 -10 23340 1592 1148 S    0  0.0  53:06.54 ovs-vswitchd
17535 root       20   0 36092 1572 1240 S    0  0.0   0:00.01 sudo
    1 root       20   0 24732 2496 1304 S    0  0.1   0:10.01 init
    2 root       20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root       20   0     0    0    0 S    0  0.0  23:43.61 ksoftirqd/0
    4 root       20   0     0    0    0 S    0  0.0   0:00.00 kworker/0:0
    5 root       20   0     0    0    0 S    0  0.0  23:19.80 kworker/u:0
    6 root       RT   0     0    0    0 S    0  0.0   3:27.44 migration/0
    7 root       RT   0     0    0    0 S    0  0.0   0:14.59 watchdog/0
```

nice

Each process has a priority setting that Linux uses to determine how CPU timeslices are shared. You can set the priority of a process by starting it with the nice command:

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

Since 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
```

Text processing

Redirection revisited

Earlier in this tutorial series, we saw an example of how to 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 -
```

Woo hoo! 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 (and requires its input to be sorted, by the way,) 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 only prints the number of lines in its input, instead of also including words and characters. You'll see that this pipeline will print out the number of unique (non-identical) lines in a text file.

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 it's 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 :)

- ***sed*** is a lightweight stream editor that can perform edits to data it receives from stdin, such as from a pipeline. Because data can just as easily be piped to sed, it's very easy to use sed as part of a long, complex pipeline in a powerful shell script.

  One of sed's most useful commands is substitution. Using it, we can replace a particular string or matched regular expression with another string. Here's an example:

```
$ sed -i
"s/#net.ipv4.conf.all.rp_filter=1/net.ipv4.conf.all.rp_filter=0/g"
/etc/sysctl.conf
```

- ***awk*** is a handy line-oriented text-processing language.
  One of awk uses is to return a specific part of another command output. Here's an example listing IP addresses on a system:

  ```
  $ ip a | awk '/ inet / { print $2 }'
  127.0.0.1/8
  10.208.148.93/17
  192.168.122.1/24
  ```

- ***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*** will print the contents of the files specified as arguments to the terminal. Handy as the first command of a pipeline, for example, cat foo.txt | blah.
- ***sort*** will print 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*** takes an already-sorted file or stream of data (via a pipeline) and removes duplicate lines.
- ***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*** 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*** is like cat, but prints all lines in reverse order; in other words, the last line is printed first.
- ***expand*** converts input tabs to spaces. Use the -t option to specify the tabstop.
- unexpand converts input spaces to tabs. Use the -t option to specify the tabstop.
- ***cut*** is used to extract character-delimited fields from each line of an input file or stream.
- The ***nl*** command adds a line number to every line of input. Useful for printouts.
- ***pr*** is used to break files into multiple pages of output; typically used for printing.
- ***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.
- ***od*** is designed to transform the input stream into a octal or hex "dump" format.
- ***split*** is a command used to split a larger file into many smaller-sized, more manageable chunks.
- ***fmt*** will reformat paragraphs so that wrapping is done at the margin. These days it's less useful since this ability is built into most text editors, but it's still a good one to know.
- ***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* is similar to paste, but it uses a field (by default the first) in each input line to match up what should be combined on a single line.
- *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 only work 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 sentinal value. This is easiest shown through an example:

```
$ sort <<EOF
apple
cranberry
banana
EOF
apple
banana
cranberry
```

In the example above, we typed the words apple, cranberry and banana, followed by "EOF" 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! What we meant was this:

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

Much better!

Kernel Modules

Meet "uname"

The uname command provides a variety of interesting information about your system. Here's what is displayed on my development workstation when I type uname -a which tells the uname command to print out all of its information in one swoop:

```
$ uname -a
Linux controller 3.2.0-49-virtual #75-Ubuntu SMP Tue Jun 18 17:59:38 UTC 2013
x86_64 x86_64 x86_64 GNU/Linux
```

More uname madness

Now, let's look at the information that uname provides

```
info. option            arg     example
kernel name             -s      "Linux"
hostname                -n      "controller"
kernel release          -r      "3.2.0-49-virtual"
kernel version          -v      "#75-Ubuntu SMP Tue Jun 18 17:59:38 UTC 2013"
machine                 -m      " x86_64"
processor               -p      " x86_64"
hardware platform       -i      " x86_64"
operating system        -o      "GNU/Linux"
```

Intriguing! What does your uname -a command print out?

The kernel release

Here's a magic trick. First, type uname -r to have the uname command print out the release of the Linux kernel that's currently running.

Now, look in the **/lib/modules** directory and --presto!-- I bet you'll find a directory with that exact name! OK, not quite magic, but now may be a good time to talk about the significance of the directories in **/lib/modules** and explain what kernel modules are.

The kernel

The Linux kernel is the heart of what is commonly referred to as "Linux" -- it's the piece of code that accesses your hardware directly and provides abstractions so that regular old programs can run. Thanks to the kernel, your text editor doesn't need to worry about whether it is writing to a SCSI or IDE disk -- or even a RAM disk. It just writes to a filesystem, and the kernel takes care of the rest.

Introducing kernel modules

So, what are kernel modules? Well, they're parts of the kernel that have been stored in a special format on disk. At your command, they can be loaded into the running kernel and provide additional functionality.

Because the kernel modules are loaded on demand, you can have your kernel support a lot of additional functionality that you may not ordinarily want to be enabled. But once in a blue moon, those kernel modules are likely to come in quite handy and can be loaded -- often automatically -- to support that odd filesystem or hardware device that you rarely use.

Kernel modules in a nutshell

In sum, kernel modules allow for the running kernel to enable capabilities on an on-demand basis. Without kernel modules, you'd have to compile a completely new kernel and reboot in order for it to support something new.

lsmod

To see what modules are currently loaded on your system, use the lsmod command:

```
# lsmod
Module                  Size  Used by
xt_conntrack           12760  0
nf_conntrack_ipv6      13906  0
nf_defrag_ipv6         13412  1 nf_conntrack_ipv6
xt_mac                 12492  0
xt_physdev             12587  0
ipt_REDIRECT           12549  0
ip6table_filter        12815  1
ip6_tables             27864  1 ip6table_filter
.
.
```

Modules listing

As you can see, my system has quite a few modules loaded to operate the various devices in my system.

Then I have a bunch of modules that are used to provide support for my USB-based input devices -- namely "mousedev," "hid," "usbmouse," "input," "usb-ohci," "ehci-hcd" and "usbcore." It often makes sense to configure your kernel to provide USB support as modules. Why? Because USB devices are "plug and play," and when you have your USB support in modules, then you can go out and buy a new USB device, plug it in to your system, and have the system automatically load the appropriate modules to enable that device.

Third-party modules

It should be noted that some of my kernel modules come from the kernel sources themselves. For example, all the USB-related modules are compiled from the standard Linux kernel sources. However, the nvidia, emu10k1 and VMWare-related modules come from other sources. This highlights another major benefit of kernel modules -- allowing third parties to provide much-needed kernel functionality and allowing this functionality to "plug in" to a running Linux kernel. No reboot necessary.

depmod and friends

In my **/lib/modules/3.2.0-49-virtual/** directory, I have a number of files that start with the string "modules.":

```
$ ls /lib/modules/2.4.20-gaming-r1/modules.*
/lib/modules/3.2.0-49-virtual/modules.alias
/lib/modules/3.2.0-49-virtual/modules.alias.bin
/lib/modules/3.2.0-49-virtual/modules.builtin
/lib/modules/3.2.0-49-virtual/modules.builtin.bin
/lib/modules/3.2.0-49-virtual/modules.ccwmap
/lib/modules/3.2.0-49-virtual/modules.dep
/lib/modules/3.2.0-49-virtual/modules.dep.bin
/lib/modules/3.2.0-49-virtual/modules.devname
/lib/modules/3.2.0-49-virtual/modules.ieee1394map
/lib/modules/3.2.0-49-virtual/modules.inputmap
/lib/modules/3.2.0-49-virtual/modules.isapnpmap
/lib/modules/3.2.0-49-virtual/modules.ofmap
/lib/modules/3.2.0-49-virtual/modules.order
/lib/modules/3.2.0-49-virtual/modules.pcimap
/lib/modules/3.2.0-49-virtual/modules.seriomap
/lib/modules/3.2.0-49-virtual/modules.softdep
/lib/modules/3.2.0-49-virtual/modules.symbols
/lib/modules/3.2.0-49-virtual/modules.symbols.bin
/lib/modules/3.2.0-49-virtual/modules.usbmap
```

These files contain some lots of dependency information. For one, they record *dependency* information for modules -- some modules require other modules to be loaded first before they will run. This information is recorded in these files.

How you get modules

Some kernel modules are designed to work with specific hardware devices. For these types of modules, these files also record the PCI IDs and similar identifying marks of the hardware devices that they support. This information can be used by things like the "hotplug" scripts to auto-detect hardware and load the appropriate module to support said hardware automatically.

Using depmod

If you ever install a new module, this dependency information may become out of date. To make it fresh again, simply type depmod -a. The depmod program will then scan all the modules in your directories in **/lib/modules** and freshen the dependency information. It does this by scanning the module files in **/lib/modules** and looking at what are called "symbols" inside the modules.

Locating kernel modules

So, what do kernel modules look like? To get a list, type the following:

```
# ls -l /lib/modules/$(uname -r)
total 580
lrwxrwxrwx 1 root root     39 Jun 18 18:41 build -> /usr/src/linux-headers-
3.2.0-49-virtual
drwxr-xr-x 2 root root   4096 Jul 12 20:27 initrd
drwxr-xr-x 9 root root   4096 Jul 12 20:27 kernel
-rw-r--r-- 1 root root  28225 Sep 17 21:56 modules.alias
-rw-r--r-- 1 root root  33490 Sep 17 21:56 modules.alias.bin
-rw-r--r-- 1 root root   5765 Jun 18 18:39 modules.builtin
-rw-r--r-- 1 root root   7159 Sep 17 21:56 modules.builtin.bin
-rw-r--r-- 1 root root     69 Sep 17 21:56 modules.ccwmap
-rw-r--r-- 1 root root  40220 Sep 17 21:56 modules.dep
-rw-r--r-- 1 root root  62790 Sep 17 21:56 modules.dep.bin
-rw-r--r-- 1 root root    186 Sep 17 21:56 modules.devname
-rw-r--r-- 1 root root     73 Sep 17 21:56 modules.ieee1394map
-rw-r--r-- 1 root root    218 Sep 17 21:56 modules.inputmap
-rw-r--r-- 1 root root    312 Sep 17 21:56 modules.isapnpmap
-rw-r--r-- 1 root root     74 Sep 17 21:56 modules.ofmap
-rw-r--r-- 1 root root 128931 Jun 18 18:39 modules.order
-rw-r--r-- 1 root root  24214 Sep 17 21:56 modules.pcimap
-rw-r--r-- 1 root root    127 Sep 17 21:56 modules.seriomap
-rw-r--r-- 1 root root    131 Sep 17 21:56 modules.softdep
-rw-r--r-- 1 root root  78332 Sep 17 21:56 modules.symbols
-rw-r--r-- 1 root root  96019 Sep 17 21:56 modules.symbols.bin
-rw-r--r-- 1 root root   6162 Sep 17 21:56 modules.usbmap
drwxr-xr-x 3 root root   4096 Aug 22 04:49 updates
```

Or to locate the directory, type this:

```
# ls -d /lib/modules/$(uname -r)
/lib/modules/3.2.0-49-virtual
```

insmod vs. modprobe

So, how does one load a module into a running kernel? One way is to use the insmod command and specifying the full path to the module that you wish to load:

```
# insmod /lib/modules/2.4.20-gaming-r1/kernel/fs/fat/fat.o
# lsmod | grep fat
fat                     29272   0   (unused)
```

However, one normally loads modules by using the modprobe command. One of the nice things about the modprobe command is that it automatically takes care of loading any dependent modules. Also, one doesn't need to specify the path to the module you wish to load, nor does one specify the trailing ".o".

rmmod and modprobe in action

Let's unload our "fat.o" module and load it using modprobe:

```
# rmmod fat
# lsmod | grep fat
# modprobe fat
# lsmod | grep fat
fat                     29272   0  (unused)
```

As you can see, the rmmod command works similarly to modprobe, but has the opposite effect -- it unloads the module you specify.

Your friend modinfo and modules

You can use the modinfo command to learn interesting things about your favorite modules:

```
# modinfo lp
filename:       /lib/modules/3.2.0-49-virtual/kernel/drivers/char/lp.ko
license:        GPL
alias:          char-major-6-*
srcversion:     27AEFDE12562FE797EC95EE
depends:        parport
intree:         Y
vermagic:       3.2.0-49-virtual SMP mod_unload modversions
parm:           parport:array of charp
parm:           reset:bool
```

And make special note of the **/etc/modules** file. This file contains configuration information for modprobe. It allows you to tweak the functionality of modprobe by telling it to load modules before/after loading others, run scripts before/after modules load, and more.

Intermediate Administration

This section covers a variety of topics, including system and Internet documentation, the Linux permissions model, user account management, and login environment tuning.

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 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
```

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 that 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

The apropos command is 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 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/manpath.config** 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.

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.

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
3560
```

The Linux Documentation Project

In addition to system documentation, there are a number of excellent Linux resources on the Internet. The Linux Documentation Project (http://www.tldp.org/) 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.

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 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- 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 root 959120 Mar 28 18:02 /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 log in 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 joe
$ whoami
joe
```

What groups am I in?

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

```
$ groups
joe wheel audio
```

In the above example, I'm a member of the joe, 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
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 joe /home/joe
```

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 script1.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 and group and for anyone else.

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

```
$ chmod -x script1.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 script1.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 script1.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 script1.sh
```

Numeric modes

Up until now, we've used what are called symbolic modes to specify permission changes to chmod. However, there's another common 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 sets the "owner", "group", and "other" flags that we've been discussing in this section. The 1 is used to set the special permissions bits, which we'll cover later (see " The elusive first digit" 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 script1.sh
$ ls -l script1.sh
-rwxr-xr-x    1 joe joe        0 Jan  9 17:34 script1.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.

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 you may not know 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 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.

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

```
$ ls -l /etc/passwd
-rw-r--r-- 1 root root 1643 Aug 22 05:41 /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 root 42824 Sep 12  2012 /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 sgid 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/someapp
```

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/joe
```

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
```

Linux account managment

Introducing /etc/passwd

In this section, we'll look at the Linux account management mechanism, starting with 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:

```
joe:x:1000:1000:Joe Joseph:/home/joe:/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 (joe), and the second field contains an x. On ancient Linux systems, this field contained 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**:

```
joe:$6$GilASJUh$Bxezod6fJ8zHpHZOvLranpT0UFwi19x/Adi1yPazpSyTXKAcDzNDYMMgNKEJs
cyeSUC.LXvFwvAUfGpZGMUjx0:15939:0:99999:7:::
```

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	# of days since 1/1/1970 that the password was modified
field 4	# of days before 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:

```
joe: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 joe user in the joe group, even though the joe 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
# 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: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 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 (one that has a password and is longer than the standard system account entries):

```
joe:$6$GilASJUh$Bxezod6fJ8zHpHZOvLranpT0UFwi19x/Adi1yPazpSyTXKAcDzNDYMMgNKEJs
cyeSUC.LXvFwvAUfGpZGMUjx0:15939:0:99999:7:::
```

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:$6$GilASJUh$Bxezod6fJ8zHpHZOvLranpT0UFwi19x/Adi1yPazpSyTXKAcDzNDYMMg
NKEJscyeSUC.LXvFwvAUfGpZGMUjx0:15939:0:99999:7:::
```

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, let's review the various time-saving account administration utilities available under Linux. Due to space constraints, we won't cover 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.

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 thisgroup 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
useradd/userdel/usermod	Used to add/delete/modify users in /etc/passwd. These commands also perform various other convenience functions. See the 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.

Tuning the user environment

Introducing "fortune"

Your shell has many useful options that you can 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 whether 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 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.

Advanced administration

In this section, we'll bolster your knowledge of advanced Linux administration skills by covering a variety of topics including Linux filesystems, the Linux boot process, runlevels, filesystem quotas, and system logs.

Filesystems, partitions, and block devices

Introduction to block devices

In this section, we'll take a good look at disk-oriented aspects of Linux, including Linux filesystems, partitions, and block devices. Once you're familar with the ins and outs of disks and filesystems, we'll guide you through the process of setting up partitions and filesystems on Linux.

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
```

Layers of abstraction

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.

Partitions

Under Linux, we create filesystems by using a special command called mkfs (or mke2fs, mkreiserfs, etc.), 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 partititons. 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.

Alternate interfaces to the disk's partition table include cfdisk, parted, and partimage. I recommend that you avoid using cfdisk (despite what the fdisk manual page may say) because it sometimes calculates disk geometry incorrectly.

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

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.

Inside 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 (each with a corresponding partition listed as "Linux") as well as a swap partition (listed as "Linux swap").

Block device and partitioning overview

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 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.

Partitioning overview, continued

All partitions hda5 and higher are logical partitions. The numbers 1 through 4 are reserved for primary or extended partitions.

In our example, hda1 through hda3 are primary partitions. hda4 is an extended partition that contains logical partitions hda5 through hda9. You would never actually use **/dev/hda4** for storing any filesystems directly -- it simply acts as a container for partitions hda5 through hda9.

Partition types

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 fileystems and swap devices on the disk at boot-time.

Using fdisk to set up partitions

Now that you've had your introduction to the way disk partitioning is done under Linux, it's time to walk through the process of setting up disk partitions and filesystems for a new Linux installation. In this process, we will configure a disk with new partitions and then create filesystems on them. These steps will provide us with a completely clean disk with no data on it that can then be used as a basis for a new Linux installation.

To follow these steps, you need to have a hard drive that does not contain any important data, since these steps will erase the data on your disk. If this is all new to you, you may want to consider just reading the steps, or using a Linux boot disk on a test system so that no data will be at risk.

What the partitioned disk will look like

After we walk through the process of creating partitions on your disk, your partition configuration will look like this:

```
Disk /dev/hda: 30.0 GB, 30005821440 bytes
240 heads, 63 sectors/track, 3876 cylinders
Units = cylinders of 15120 * 512 = 7741440 bytes

   Device Boot    Start       End    Blocks   Id  System
/dev/hda1   *         1        14    105808+  83  Linux
/dev/hda2            15        81    506520   82  Linux swap
/dev/hda3            82      3876  28690200   83  Linux

Command (m for help):
```

Sample partition commentary

In our suggested "newbie" partition configuration, we have three partitions. The first one (**/dev/hda1**) at the beginning of the disk is a small partition called a boot partition. The boot partition's purpose is to hold all the critical data related to booting -- GRUB boot loader information (if you will be using GRUB) as well as your Linux kernel(s). The boot partition gives us a safe place to store everything related to booting Linux. During normal day-to-day Linux use, your boot partition should remain unmounted for safety. If you are setting up a SCSI system, your boot partition will likely end up being **/dev/sda1**.

It's recommended to have boot partitions (containing everything necessary for the boot loader to work) at the beginning of the disk. While not necessarily required anymore, it is a useful tradition from the days when the LILO boot loader wasn't able to load kernels from filesystems that extended beyond disk cylinder 1024.

The second partition (**/dev/hda2**) is used for swap space. The kernel uses swap space as virtual memory when RAM becomes low. This partition, relatively speaking, isn't very big either, typically somewhere around 512 MB. If you're setting up a SCSI system, this partition will likely end up being called **/dev/sda2**.

The third partition (**/dev/hda3**) is quite large and takes up the rest of the disk. This partition is called our root partition and will be used to store your main filesystem that houses the main Linux filesystem. On a SCSI system, this partition would likely end up being **/dev/sda3**.

Getting started

Okay, now to create the partitions as in the example and table above. First, enter fdisk by typing fdisk /dev/hda or fdisk /dev/sda, depending on whether you're using IDE or SCSI. Then, type p to view your current partition configuration. Is there anything on the disk that you need to keep? If so, stop now. If you continue with these directions, all existing data on your disk will be erased.

Following the instructions below will cause all prior data on your disk to be erased! If there is anything on your drive, please be sure that it is non-critical information that you don't mind losing. Also make sure that you have selected the correct drive so that you don't mistakenly wipe data from the wrong drive.

Zapping existing partitions

Now, it's time to delete any existing partitions. To do this, type d and hit enter. You will then be prompted for the partition number you would like to delete. To delete a pre-existing **/dev/hda1**, you would type:

```
Command (m for help): d
Partition number (1-4): 1
```

The partition has been scheduled for deletion. It will no longer show up if you type p, but it will not be erased until your changes have been saved. If you made a mistake and want to abort without saving your changes, type q immediately and hit enter and your partition will not be deleted.

Now, assuming that you do indeed want to wipe out all the partitions on your system, repeatedly type p to print out a partition listing and then type d and the number of the partition to delete it. Eventually, you'll end up with a partition table with nothing in it:

```
Disk /dev/hda: 30.0 GB, 30005821440 bytes
240 heads, 63 sectors/track, 3876 cylinders
Units = cylinders of 15120 * 512 = 7741440 bytes

Device Boot    Start        End     Blocks    Id  System

Command (m for help):
```

Creating a boot partition

Now that the in-memory partition table is empty, we're ready to create a boot partition. To do this, type n to create a new partition, then p to tell fdisk you want a primary partition. Then type 1 to create the first primary partition. When prompted for the first cylinder, hit enter. When prompted for the last cylinder, type +100M to create a partition 100MB in size. Here's the output from these steps:

```
Command (m for help): n
Command action
e   extended
p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-3876, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-3876, default 3876): +100M
```

Now, when you type p, you should see the following partition printout:

```
Command (m for help): p

Disk /dev/hda: 30.0 GB, 30005821440 bytes
240 heads, 63 sectors/track, 3876 cylinders
Units = cylinders of 15120 * 512 = 7741440 bytes

   Device Boot    Start       End    Blocks   Id  System
/dev/hda1             1        14    105808+  83  Linux
```

Creating the swap partition

Next, let's create the swap partition. To do this, type n to create a new partition, then p to tell fdisk that you want a primary partition. Then type 2 to create the second primary partition,**/dev/hda2** in our case. When prompted for the first cylinder, hit enter. When prompted for the last cylinder, type +512M to create a partition 512MB in size. After you've done this, type t to set the partition type, and then type 82 to set the partition type to "Linux Swap." After completing these steps, typing p should display a partition table that looks similar to this:

```
Command (m for help): p

Disk /dev/hda: 30.0 GB, 30005821440 bytes
240 heads, 63 sectors/track, 3876 cylinders
Units = cylinders of 15120 * 512 = 7741440 bytes

   Device Boot    Start       End    Blocks   Id  System
/dev/hda1             1        14    105808+  83  Linux
/dev/hda2            15        81    506520   82  Linux swap
```

Making it bootable

Finally, we need to set the "bootable" flag on our boot partition and then write our changes to disk. To tag **/dev/hda1** as a "bootable" partition, type a at the menu and then type 1 for the partition number. If you type p now, you'll now see that /dev/hda1 has an "*" in the "Boot" column. Now, let's write our changes to disk. To do this, type w and hit enter. Your disk partitions are now properly configured for the installation of Linux.

If fdisk instructs you to do so, please reboot to allow your system to detect the new partition configuration.

Extended and logical partitioning

In the above example, we created a single primary partition that will contain a filesystem used to store all our data. This means that after installing Linux, this main filesystem will get mounted at "/" and will contain a tree of directories that contain all our files.

While this is a common way to set up a Linux system, there is another approach that you should be familiar with. This approach uses multiple partitions that house multiple filesystems and are then "linked" together to form a cohesive filesystem tree. For example, it is common to put **/home** and **/var** on their own filesystems.

We could have made hda2 into an extended rather than a primary partition. Then, we could have created the hda5, hda6, and hda7 logical partitions (which would technically be contained "inside" hda2), which would house the **/**, **/home**, and **/var** filesystems respectively.

You can learn more about these types of multi-filesystem configurations by studying the resources listed on the next page.

Creating filesystems

Now that the partitions have been created, it's time to set up filesystems on the boot and root partitions so that they can be mounted and used to store data. We will also configure the swap partition to serve as swap storage.

Linux supports a variety of different types of filesystems; each type has its strengths and weaknesses and its own set of performance characteristics. We will cover the creation of ext2, ext3, XFS, JFS, and ReiserFS filesystems in this tutorial. Before we create filesystems on our example system, let's briefly review the various filesystems available under Linux. We'll go into more detail on the filesystems later in the tutorial.

The ext2 filesystem

ext2 is the tried-and-true Linux filesystem, but it doesn't have metadata journaling, which means that routine ext2 filesystem checks at startup time can be quite time-consuming. There is now quite a selection of newer-generation journaled filesystems that can be checked for consistency very quickly and are thus generally preferred over their non-journaled counterparts. Journaled filesystems prevent long delays when you boot your system and your filesystem happens to be in an inconsistent state.

The ext3 filesystem

ext3 is the journaled version of the ext2 filesystem, providing metadata journaling for fast recovery in addition to other enhanced journaling modes, such as full data and ordered data journaling. ext3 is a very good and reliable filesystem. It offers generally decent performance under most conditions. Because it does not extensively employ the use of "trees" in its internal design, it doesn't scale very well, meaning that it is not an ideal choice for very large filesystems or situations where you will be handling very large files or large quantities of files in a single directory. But when used within its design parameters, ext3 is an excellent filesystem.

One of the nice things about ext3 is that an existing ext2 filesystem can be upgraded "in-place" to ext3 quite easily. This allows for a seamless upgrade path for existing Linux systems that are already using ext2.

The ReiserFS filesystem

ReiserFS is a B-tree-based filesystem that has very good overall performance and greatly outperforms both ext2 and ext3 when dealing with small files (files less than 4k), often by a factor of 10x-15x. ReiserFS also scales extremely well and has metadata journaling. As of kernel 2.4.18+, ReiserFS is now rock-solid and highly recommended for use both as a general-purpose filesystem and for extreme cases such as the creation of large filesystems, the use of many small files, very large files, and directories containing tens of thousands of files. ReiserFS is the filesystem we recommend by default for all non-boot partitions.

The XFS filesystem

XFS is a filesystem with metadata journaling. It comes with a robust feature-set and is optimized for scalability. We only recommend using this filesystem on Linux systems with high-end SCSI and/or fibre channel storage and a uninterruptible power supply. Because XFS aggressively caches in-transit data in RAM, improperly designed programs (those that don't take proper precautions when writing files to disk (and there are quite a few of them) can lose a good deal of data if the system goes down unexpectedly.

The JFS filesystem

JFS is IBM's own high performance journaling filesystem. It has recently become production-ready, and we would like to see a longer track record before commenting either positively nor negatively on its general stability at this point.

Filesystem recommendations

If you're looking for the most rugged journaling filesystem, use ext3. If you're looking for a good general-purpose high-performance filesystem with journaling support, use ReiserFS; both ext3 and ReiserFS are mature, refined and recommended for general use.

Based on our example above, we will use the following commands to initialize all our partitions for use:

```
# mke2fs -j /dev/hda1''
# mkswap /dev/hda2
# mkreiserfs /dev/hda3
```

We choose ext3 for our **/dev/hda1** boot partition because it is a robust journaling filesystem supported by all major boot loaders. We used mkswap for our **/dev/hda2** swap partition -- the choice is obvious here. And for our main root filesystem on **/dev/hda3** we choose ReiserFS, since it is a solid journaling filesystem offering excellent performance. Now, go ahead and initialize your partitions.

Making swap

mkswap is the command that used to initialize swap partitions:

```
# mkswap /dev/hda2
```

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

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

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.

Creating ext2, ext3, and ReiserFS filesystems

You can use the mke2fs command to create ext2 filesystems:

```
# mke2fs /dev/hda1
```

If you would like to use ext3, you can create ext3 filesystems using mke2fs -j:

```
# mke2fs -j /dev/hda3
```

To create ReiserFS filesystems, use the mkreiserfs command:

```
# mkreiserfs /dev/hda3
```

Creating XFS and JFS filesystems

To create an XFS filesystem, use the mkfs.xfs command:

```
# mkfs.xfs /dev/hda3
```

You may want to add a couple of additional flags to the mkfs.xfs command: '-d agcount=3 -l size=32m'. The '-d agcount=3' command will lower the number of allocation groups. XFS will insist on using at least one allocation group per 4GB of your partition, so, for example, if you have a 20GB partition you will need a minimum agcount of 5. The '-l size=32m' command increases the journal size to 32MB, increasing performance.

To create JFS filesystems, use the mkfs.jfs command:

```
# mkfs.jfs /dev/hda3
```

Mounting filesystems

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

```
# mount /dev/hda3 /mnt/custom
```

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/custom** directory on the parent filesystem. Later, when the filesystem is unmounted, these files will reappear. After executing the mount command, any files created or copied inside **/mnt/custom** will be stored on the new ReiserFS filesystem you mounted.

Let's say we wanted to mount our boot partition inside **/mnt/custom**. We could do this by performing the following steps:

```
# mkdir /mnt/custom/boot
# mount /dev/hda1 /mnt/custom/boot
```

Now, our boot filesystem is available inside **/mnt/custom/boot**. If we create files inside **/mnt/custom/boot**, they will be stored on our ext3 filesystem that physically resides on **/dev/hda1**. If we create file inside **/mnt/custom** but not **/mnt/custom/boot**, then they will be stored on our ReiserFS filesystem that resides on **/dev/hda3**. And if we create files outside of **/mnt/custom**, they will not be stored on either filesystem but on the filesystem of our current Linux system or boot disk.

To see what filesystems are mounted, type mount by itself. Here is the output of mount on one of our currently-running Linux system, which has partitions configured identically to those in the example above:

```
/dev/root on / type reiserfs (rw,noatime)
none on /dev type devfs (rw)
proc on /proc type proc (rw)
tmpfs on /dev/shm type tmpfs (rw)
usbdevfs on /proc/bus/usb type usbdevfs (rw)
/dev/hde1 on /boot type ext3 (rw,noatime)
```

You can also view similar information by typing cat /proc/mounts. The "root" filesystem, **/dev/hda3** gets mounted automatically by the kernel at boot-time, and gets the symbolic name **/dev/hda3**. On our system, both **/dev/hda3** and **/dev/root** point to the same underlying block device using a symbolic link:

```
# ls -l /dev/root
lr-xr-xr-x   1 root   root   33 Mar 26 20:39 /dev/root ->
ide/host0/bus0/target0/lun0/part3

# ls -l /dev/hda3
lr-xr-xr-x   1 root   root   33 Mar 26 20:39 /dev/hde3 ->
ide/host0/bus0/target0/lun0/part3
```

Even more mounting stuff

So, what is this "**/dev/ide/host0....**" file? Systems like mine that use the 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.

When using the mount command to mount filesystems, it attempts to auto-detect the filesystem type. Sometimes, this may not work and you will need to specify the to-be-mounted filesystem type manually using the -t option, as follows:

```
# mount /dev/hda1 /mnt/boot -t ext3
```

or

```
# mount /dev/hda3 /mnt -t reiserfs
```

Mount options

It's also 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/custom -o ro
```

With **/dev/hdc6** mounted read-only, no files can be modified in **/mnt/custom** -- 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/custom -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/custom** is associated with **/dev/hdc6**. To make the filesystem writeable again, we can remount it as read-write:

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

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

Introducing fstab

So far, we've seen how partition an example disk and mount filesystems manually from a boot disk. But once we get a Linux system installed, how do we configure that Linux system to mount the right filesystems at the right time? For example, Let's say that we installed Gentoo Linux on our current example filesystem configuration. How would our system know how to to find the root filesystem on **/dev/hda3**? And if any other filesystems -- like swap -- needed to be mounted at boot time, how would it know which ones?

Well, the Linux kernel is told what root filesystem to use by the boot loader, and we'll take a look at the linux boot loaders later in this tutorial. But for everything else, your Linux system has a file called **/etc/fstab** that tells it about what filesystems are available for mounting. Let's take a look at it.

A sample fstab

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

```
  <fs> <mountpoint> <type>    <opts>            <dump/pass>

/dev/hda1       /boot           ext3            noauto,noatime          1 1
/dev/hda3       /               reiserfs        noatime                 0 0
/dev/hda2       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 ext3 filesystem that should be mounted at the /boot mountpoint. Now, look at **/dev/hda1'**s 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.

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/hda2**. This line defines **/dev/hda2** 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/hda2** 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/custom
```

or

```
# umount /dev/hda3
```

Once unmounted, any files in **/mnt/custom** 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**.

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. For some journaling filesystems like ReiserFS, it is safe to have a passno of 0 since the journaling code (and not an external fsck) takes care of making the filesystem consistent again.

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 its 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.

Let's cover the major filesystems available for Linux, along with their associated commands and options.

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 fsck 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.

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
- 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.)

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. In general, ReiserFS offers very good performance in most all situations. ReiserFS is preferred by many for its speed and scalability.

- In kernels: 2.4.0+ (2.4.18+ strongly recommended)
- journaling: metadata
- mkfs command: mkreiserfs
- mkfs example: mkreiserfs /dev/hdc7
- performance-related mount options: noatime, notail

The XFS filesystem

The XFS filesystem is an enterprise-class journaling filesystem being ported to Linux by SGI. XFS is a full-featured, scalable, journaled file-system that is a good match for high-end, reliable hardware (since it relies heavily on caching data in RAM.) but not a good match for low-end hardware.

- In kernels: 2.5.34+ only, requires patch for 2.4 series
- journaling: metadata
- mkfs command: mkfs.xfs
- mkfs example: mkfs.xfs /dev/hdc7
- performance-related mount options: noatime
- XFS Resources:

The JFS filesystem

JFS is a high-performance journaling filesystem ported to Linux by IBM. JFS is used by IBM enterprise servers and is designed for high-performance applications. You can learn more about JFS at the JFS project Web site.

- In kernels: 2.4.20+
- journaling: metadata
- mkfs command: mkfs.jfs
- mkfs example: mkfs.jfs /dev/hdc7
- performance-related mount options: noatime
- JFS Resources:

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.

Booting the system

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 of which distribution is installed. Consider the following example hard disk:

```
+-----------------+
|       MBR       |
+-----------------+
|  Partition 1:   |
| Linux root (/)  |
|   containing    |
|   kernel and    |
|     system.     |
+-----------------+
|  Partition 2:   |
|   Linux swap    |
+-----------------+
|  Partition 3:   |
|   Windows 3.0   |
|  (last booted   |
|    in 1992)     |
+-----------------+
```

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:

```
[    0.000000] Linux version 3.2.0-49-virtual (buildd@komainu) (gcc version
4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5) ) #75-Ubuntu SMP Tue Jun 18 17:59:38 UTC
2013 (Ubuntu 3.2.0-49.75-virtual 3.2.46)
```

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-legacy

The GRUB-legacy boot loader is another popular Linux boot loader. GRUB-legacy supports more operating systems than LILO, provides some password-based security in the boot menu, and is easier to administer.

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

Using GRUB-legacy

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.

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:

```
[    0.000000] Linux version 3.2.0-49-virtual (buildd@komainu) (gcc version
4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5) ) #75-Ubuntu SMP Tue Jun 18 17:59:38 UTC
2013 (Ubuntu 3.2.0-49.75-virtual 3.2.46)
```

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#53
Jan 12 20:17:41 time /usr/sbin/named[350]: running
```

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.

Understanding 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.

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 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 shut down or start up 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!
```

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.

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.

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.

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
```

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 600MB 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.

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 (again, 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 filesytem. 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 of 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 his or her 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 seven 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 we 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
```

System logs

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 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.conf'
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#53
Jan 12 20:17:41 bilbo /usr/sbin/named[350]: listening on IPv4 interface eth0,
10.0.0.1#53
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 may 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:

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 featureful 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.