Think Links

By Jerry Peek

Why would you want to give a program more than one name? How can you move quickly through the filesystem like *Star Trek's* Enterprise jumping through a "worm hole"? What good are multiple views of the files in a directory? You'll see these things and more, as we look into Linux filesystem *links*.

A link is similar to a *shortcut* on a *Microsoft Windows* desktop or an *alias* on the *Macintosh:* the link symbolizes a file or folder that's located somewhere else in the filesystem. In most cases, operations on the link (such as open, read, and write), affect the file the link represents, not the link itself.

For example, the sequence of commands...

```
$ touch /tmp/reminders
$ ln -s /tmp/reminders ticklers
$ echo "Send story to my editor" > ticklers
$ cat /tmp/reminders
$ cat ticklers
```

... appends text to */tmp/reminders*. Since *ticklers* is a link, all read and write operations on *ticklers* affect what *ticklers* points to, not *ticklers* itself. Links provide convenience and abstraction.

Linux filesystems have two kinds of links. And, as you'd expect from a versatile system like Linux, you can use links in some surprising ways.

Our first few sections introduce the concepts behind links. Understanding some of this can take a little time and thought, but once you've got it, you'll find all kinds of uses for links! So, let's dig in.

Two Kinds of Links

Linux has two kinds of filesystem links: *symbolic* links and *hard* links.

A symbolic link — also called a *soft link* or *symlink* — resembles a Windows shortcut. A symlink is a little file that contains the pathname of another object on the filesystem: a file, a directory, a socket, and so on — possibly even the pathname of another link. This pathname can be absolute or relative.

A hard link isn't itself a file. Instead, it's a directory entry. It points to another file using the file's *inode number*. (To learn more about *inodes*, see "Journaling File Systems" in the October 2002 issue of *Linux Magazine*, available online at http://www.linux-mag.com/2002-10/jfs_01.html.)

To give a file more than one name or to make the same file appear in multiple directories, you can make links to that file instead of copying it. One advantage of this is that a link takes little or even no disk space. Another is that, if you edit the target of the link, those changes can be seen immediately through the link.

For instance, if your link named *today* points to a target named *2004-09-01*, and you edit *2004-09-01*, those edits will also be visible immediately to anyone who reads *today* with (for example) the command `less today` or opens it in a file browser. Tomorrow, you can also replace *today* with a link to the file *2004-09-02*, and so on.

Symbolic Links

As mentioned above, a symbolic link is a little file that contains the pathname of another filesystem object. To make a symlink, use *ln* with the `-s` option. Give the name of the target first, then the name of the link. For an example of this process, see *Listing One*.

The output of `ls -l` displays a symbolic link as type `l` (look at the the first column of output) and adds `-target` after the name of the link to show what the link points to. The size of the *today* link is 10, because the target filename *2004-09-01* has 10 characters.

What happens when you use another application (a text editor, for instance) to open a symbolic link? That depends on the application. Most applications open the target of a symlink instead of the link itself. This is probably what you want.

However, a few commands and applications, like `ls -l` and *tar*, operate on the link itself unless you tell them to find the target. To make *ls* show the target of a link, add the `-L` or `--dereference` option, like `ls -lL`. To make *tar* use the target of a link instead of the link itself, add its `-h` or `--dereference` option.

Hard Links

When Unix started, its filesystems had only hard links (and they were simply called "links"). A hard link is a directory entry that refers to the actual file on the disk. There's no "target" of a hard link.

A Linux file isn't actually "in" a directory. (For an explanation of Unix and Linux file system concepts, see the October 2002 "Power Tools" column, "What's in a Pathname?" available online at http://www.linux-mag.com/2002-10/power_01.html.) Instead, the directory contains a hard link to the file information stored in an inode or similar structure. So, when you give Linux a filename (such as *foo* or */proj/foo*), it looks at the file's (hard) link to get the file's inode number. The actual file is located by its inode number.

LISTING ONE: Making a symbolic link with `ln -s`

```
$ ls -l 2004-09-01
-rw-r--r--    1 jpeek  jpeek   6124 Sep  1 08:15 2004-09-01
$ ln -s 2004-09-01 today
$ ls -l 2004-09-01 today
-rw-r--r--    1 jpeek  jpeek   6124 Sep  1 08:15 2004-09-01
lrwxrwxrwx   1 jpeek  jpeek     10 Sep  1 08:22 today -> 2004-09-01
$ ls -lL today
-rw-r--r--    1 jpeek  jpeek   6124 Sep  1 08:15 today
```

Why does this matter? Because a hard link isn't a file, and adding another hard link to a file doesn't consume any additional disk space! (That's not quite true. A link does occupy a few bytes in the directory file, but unless adding the link causes the directory file to grow to another disk block, the link effectively doesn't consume any more space.) Moreover, a file isn't actually deleted from the disk until there are no hard links to it — that is, until its *link count* goes to zero. You can see a file's link count in the output of `ls -l`.

Listing Two shows an example of creating a hard link to a file. The *ls* option `-i` shows the file's inode number in the first column of the output

At first, there is a file (which is actually a hard link) named *2004-08-31*. The file's inode number is `54147` and it has `9822` characters. Its link count is `1`. Using *ln* (*without* the `-s` option), we add a second hard link named *yesterday*.

The second hard link shows the same file data as the first one: same inode number, same character count, and the same last-modified date. That's because both links refer to the same disk file. The only difference is that now the file's link count is `2` (because there are two hard links) and the name of the second link is different (which is required because both links are in the same directory). Compare this to the symbolic link example in *Listing One*.

Now, let's look at some examples of using links.

Morphing a File, a Directory, Or ...

The previous two sections showed you how to use a link to give a file more than one name. Given a series of files, one per day (*2004-08-31*, *2004-09-01*, and so on), you can use a link named *today* to point to today's log file and another link named *yesterday* to point to yesterday's log.

Then, each morning, you (or a *cron* job) can remove the old *today* and *yesterday* links and replace them with new links to today's logs. The link names stay *constant*, but the target of the link *changes*.

The same technique can maintain multiple versions of a program. For instance, you might download and build the bleeding-edge version of some powerful program called *prog*. However, you want to keep the most recent stable version of *prog* in case the development version has a bug. If you assign the two executables different names — for example, *prog-devel-2.4.3* and *prog-stable-2.4* for the development and stable versions, respectively — you can make a link named *prog* to whichever version you want to use today. Simply invoking *prog* resolves the link and runs the version it points to.

The same technique works for entire directory trees. For instance, two versions of *X11* (the *X Window System*) could be in the directories */usr/X11R66/* and */usr/X11R67/*. Each of those directories would contain subdirectories named *bin*, *doc*, *man*, and so on. A symlink named */usr/X11* can be used to point to either *X11R66* or *X11R67*, depending on which version of X11 you want to use.

Two commands do the trick:

```
# cd /usr
# ln -s X11R66 X11
```

As another example, let's say you have a directory */usr/local/lib/* for your library files, but a package named *prog* insists on storing its library files in */usr/local/prog/lib/*. You might move the directory and replace it with a symlink, like this:

```
# cd /usr/local/prog
# mv -i lib ../lib/prog
# ln -s ../lib/prog lib
```

Or, you might make a symlink named *prog* from */usr/local/lib/* that points to the */usr/local/prog/lib/* directory:

LISTING TWO: Making a hard link with `ln`

```
$ ls -li 2004-08-31
54147 -rw-r--r--   1 jpeek  jpeek   9822 Aug 31 16:44 2004-08-31
$ ln 2004-08-31 yesterday
$ ls -li 2004-08-31 yesterday
54147 -rw-r--r--   2 jpeek  jpeek   9822 Aug 31 16:44 2004-08-31
54147 -rw-r--r--   2 jpeek  jpeek   9822 Aug 31 16:44 yesterday
```

LISTING THREE: Shell script with several names

```
#!/bin/sh

case "$0" in
*dork)    ssh -C pj@dork.us "$@" ;;
*igor)    ssh -1 igor.boris.xyz "$@" ;;
*sunspot) ssh sunspot.xyz.edu "$@" ;;
*) echo "$0: can't run myself" &2; exit 1 ;;
esac
```

```
#  cd /usr/local/lib
#  ln -s ../prog/lib prog
```

In both of the former examples, some testing is a good idea: changing an application's directories through a symlink can break the application. The section "The Kinks of Links" (below) explains.

Why use a relative pathname like *X11R66* and *../prog/lib* as link targets instead of the more obvious, absolute pathnames like */usr/X11R66* or */usr/local/prog/lib*? One advantage of using relative pathnames within a directory tree is that you can move or copy a part of the tree and the relationships between its parts (parent, child, and sibling directories) will be maintained without remaking any symlinks within that tree. This also can be important when you access a symlink on a filesystem that's mounted remotely from another computer, because a pathname stored inside a symlink must be valid as-is on both the local and remote systems.

For instance, the filesystem mounted at */usr/local/* on the system named *boston* might also be mounted at */boston/local/* on a remote system named *chicago*. To a user on *chicago*, any symlink underneath */boston/local/* that points to */usr/local/* is pointing to *chicago*'s own */usr/local* instead of where it should point: */boston/local*! So, if any of your filesystems are mounted remotely, be careful using absolute pathnames as symlink targets.

A Program By Any Other Name...

We've seen that links can give a file more than one name. Since Linux programs are just (special) files, programs can also be given multiple names. And because a program can check the name it was invoked with, you can give the same program multiple (and usually related) purposes that change depending on the program name you use.

For instance, when you list the executable files for the GNU *zip* program, you'll see that the three commands are actually the same file:

```
% ls -li /bin/{gzip,gunzip,zcat}
89203 -rwxr-xr-x  3 .... /bin/gunzip
89203 -rwxr-xr-x  3 .... /bin/gzip
89203 -rwxr-xr-x  3 .... /bin/zcat
```

Many developers use this technique to change the personality of an application depending on how it's invoked.

Listing Three shows an example script that uses this "alias" technique. This Bourne shell script may be invoked with three different names: *dork*, *igor*, and *sunspot*. The script is used to connect to three remote machines via *ssh*, with different settings on each. The `$0` variable expands into the name the script was called with. This may be a pathname like `./dork` or `/u/jpeek/bin/igor`, so the *case* statement starts each pattern with `*` to match any leading pathname. The `"$@"` expands

LISTING FOUR: Breaking a hard link

```
$ ls -li
total 10
54147 -rw-r--r--   2 jpeek  jpeek   9822 Aug 31 16:44 link
$ mv link link.bak
$ cp link.bak link
$ ls -li
total 20
54152 -rw-r--r--   1 jpeek  jpeek   9822 Sep  1 09:24 link
54147 -rw-r--r--   2 jpeek  jpeek   9822 Aug 31 16:44
  link.bak
```

HARD LINKS TO A DIRECTORY

You can use `ln -s` to make a symbolic link to a directory. But *ln* won't make a hard link to a directory unless you're the superuser — and even then, it's dangerous to do. Why? Because a directory has a series of hard links that must be maintained carefully to avoid filesystem corruption.

When you create a new directory with *mkdir*, it makes the proper hard links automatically. Watching this happen shows you the links. Let's make a directory under */tmp* named *linktest* and experiment with it.

```
% cd /tmp
% mkdir linktest
% ls -ldi linktest
34063 drwxrwxr-x  2 jpeek  jpeek  4096 Sep  3
  12:38 linktest
% cd linktest
% ls -ldi .
34063 drwxrwxr-x  2 jpeek  jpeek  4096 Sep  3
  12:38 .
```

The link count of 2 shows that the directory has two hard links. `ls -ldi` shows them both: the directory file itself and its entry named . (which you can use while you're in any directory to refer to the directory itself). Now, let's make three subdirectories and do more listing.

```
% mkdir a b c
$ ls -lid . [abc]/..
34063 drwxrwxr-x  5 jpeek  jpeek  4096 Sep  3 12:40 .
34063 drwxrwxr-x  5 jpeek  jpeek  4096 Sep  3 12:40 a/..
34063 drwxrwxr-x  5 jpeek  jpeek  4096 Sep  3 12:40 b/..
34063 drwxrwxr-x  5 jpeek  jpeek  4096 Sep  3 12:40 c/..
```

Now the directory's link count has jumped from 2 to 5. Those three new hard links are in the subdirectories: every subdirectory has a hard link named .. to its parent.

This web of hard links is what ties the Linux filesystem together. A superuser can change it, but only at the risk of trouble. *rmdir* only removes the hard links it expects to find in a well-formed directory tree, so any others left over become orphaned and must be cleaned up with *fsck*.

into any command-line arguments. For instance, if you run the script as `igor`, it executes `ssh -1 igor.boris.xyz` to log into the host *igor.boris.xyz* using SSH version 1. Or, if you type `dork df`, the script runs `ssh -C pj@dork.us df`, which calls the *df* utility to show filesystem usage on the host *dork.us*, logging in as *pj* instead of the default, and compressing (`-C`) the connection.

This technique, which extracts the name of the command, has equivalents in many other languages, too.

The Kinks of Links

Let's finish by looking at some link "gotchas."

One gotcha happens when a program changes its current directory to, or through, a symbolic link. Because it isn't trivial to determine the current directory from scratch, some programs determine the new current directory path by starting at the previous path.

For example, if the current directory is */usr/foo* and a shell user runs `cd bar`, the shell may decide that its new current directory is at */usr/foo/bar*. But if *bar* is a symlink to *../beer*, the actual current directory will be */usr/beer!* You can check a shell for this problem by running *pwd* before and after you use *cd* — or by searching the shell's *man* page for the word *link*.

Here's one workaround for problems with linked directories. The *lndir* utility, which comes with X11, makes a shadow copy of a directory tree that's filled with symbolic links instead of the actual files. This is useful when you need more than a symbolic link to the directory: you need a separate directory where you can create other files. *lndir* is handy for building software for multiple architectures: one directory has the master source files, while other directories have symlinked source files and the built files for a particular architecture.

Another gotcha is broken links. If the target of a symlink is moved or removed, the symlink now points to "nowhere." A hard link breaks if you move (rename) a hard-linked file to another filesystem (that's because a hard link always refers to its current filesystem, so a new file must be created at the destination). A hard link can also break if a file is renamed and copied back to the original filename — when a text editor makes a backup, for instance.

Listing Four shows an example: the file named link has been replaced by a new file, and now the old file is named *link.bak*.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.