Symbolic Links

Symbolic links were created to overcome the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a Windows shortcut though of course, they predate the Windows feature by many years ;-)

A file pointed to by a symbolic link, and the symbolic link itself are largely indistinguishable from one another. For example, if you write something to the symbolic link, the referenced file is written to. However when you delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist, but will point to nothing. In this case, the link is said to be *broken*. In many implementations, the `ls` command will display broken links in a distinguishing color, such as red, to reveal their presence.

The concept of links can seem very confusing, but hang in there. We're going to try all this stuff and it will, hopefully, become clear.

Let's Build A Playground

Since we are going to do some real file manipulation, let's build a safe place to “play” with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it “playground.”

Creating Directories

The `mkdir` command is used to create a directory. To create our playground directory we will first make sure we are in our home directory and will then create the new directory:

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

To make our playground a little more interesting, let's create a couple of directories inside it called “dir1” and “dir2”. To do this, we will change our current working directory to `playground` and execute another `mkdir`:

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

Notice that the `mkdir` command will accept multiple arguments allowing us to create both directories with a single command.

Copying Files

Next, let's get some data into our playground. We'll do this by copying a file. Using the `cp` command, we'll copy the `passwd` file from the `/etc` directory to the current working directory:

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Notice how we used the shorthand for the current working directory, the single trailing period. So now if we perform an `ls`, we will see our file:

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me  me 4096 2016-01-10 16:40 dir1
drwxrwxr-x 2 me  me 4096 2016-01-10 16:40 dir2
-rw-r--r-- 1 me  me 1650 2016-01-10 16:07 passwd
```

Now, just for fun, let's repeat the copy using the “-v” option (verbose) to see what it does:

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

The `cp` command performed the copy again, but this time displayed a concise message indicating what operation it was performing. Notice that `cp` overwrote the first copy without any warning. Again this is a case of `cp` assuming that you know what you’re are doing. To get a warning, we'll include the “-i” (interactive) option:

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responding to the prompt by entering a “y” will cause the file to be overwritten, any other character (for example, “n”) will cause `cp` to leave the file alone.

Moving And Renaming Files

Now, the name “passwd” doesn't seem very playful and this is a playground, so let's change it to something else:

```
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again:

```
[me@linuxbox playground]$ mv fun dir1
```

to move it first to directory `dir1`, then:

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

to move it from `dir1` to `dir2`, then:

```
[me@linuxbox playground]$ mv dir2/fun .
```

to finally bring it back to the current working directory. Next, let's see the effect of `mv` on directories. First we will move our data file into `dir1` again:

```
[me@linuxbox playground]$ mv fun dir1
```

then move `dir1` into `dir2` and confirm it with `ls`:

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me  me   4096 20016-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me  me   1650 2016-01-10 16:33 fun
```

Note that since `dir2` already existed, `mv` moved `dir1` into `dir2`. If `dir2` had not existed, `mv` would have renamed `dir1` to `dir2`. Lastly, let's put everything back:

```
[me@linuxbox playground]$ mv dir2/dir1 .
```

```
[me@linuxbox playground]$ mv dir1/fun .
```

Creating Hard Links

Now we'll try some links. First the hard links. We'll create some links to our data file like so:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

So now we have four instances of the file "fun". Let's take a look at our playground directory:

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2016-01-14 16:17 dir1
drwxrwxr-x 2 me   me   4096 2016-01-14 16:17 dir2
-rw-r--r-- 4 me   me   1650 2016-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2016-01-10 16:33 fun-hard
```

One thing you notice is that the second field in the listing for `fun` and `fun-hard` both contain a "4" which is the number of hard links that now exist for the file. You'll remember that a file will always have at least one link because the file's name is created by a link. So, how do we know that `fun` and `fun-hard` are, in fact, the same file? In this case, `ls` is not very helpful. While we can see that `fun` and `fun-hard` are both the same size (field 5), our listing provides no way to be sure. To solve this problem, we're going to have to dig a little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts: the data part containing the file's contents and the name part which holds the file's name. When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an *inode*, which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.

The `ls` command has a way to reveal this information. It is invoked with the "-i" option:

```
[me@linuxbox playground]$ ls -li
```

```
total 16
12353539 drwxrwxr-x 2 me   me   4096 2016-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me   me   4096 2016-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me   me   1650 2016-01-10 16:33 fun
12353538 -rw-r--r-- 4 me   me   1650 2016-01-10 16:33 fun-hard
```

In this version of the listing, the first field is the inode number and, as we can see, both `fun` and `fun-hard` share the same inode number, which confirms they are the same file.

Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links: Hard links cannot span physical devices and hard links cannot reference directories, only files. Symbolic links are a special type of file that contains a text pointer to the target file or directory.

Creating symbolic links is similar to creating hard links:

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

The first example is pretty straightforward, we simply add the “-s” option to create a symbolic link rather than a hard link. But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the `ls` output:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me   me   1650 2016-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      6 2016-01-15 15:17 fun-sym -> ../fun
```

The listing for `fun-sym` in `dir1` shows that it is a symbolic link by the leading “l” in the first field and that it points to “../fun”, which is correct. Relative to the location of `fun-sym`, `fun` is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string “../fun” rather than the length of the file to which it is pointing.

When creating symbolic links, you can either use absolute pathnames:

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames, as we did in our earlier example. In most cases, using relative pathnames is more desirable because it allows a directory tree containing symbolic links and their referenced files to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories:

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2016-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir2
-rw-r--r-- 4 me   me   1650 2016-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2016-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      3 2016-01-15 15:15 fun-sym -> fun
```

Removing Files And Directories

As we covered earlier, the `rm` command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let's delete one of our hard links:

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2016-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir2
-rw-r--r-- 3 me   me   1650 2016-01-10 16:33 fun
lrwxrwxrwx 1 me   me      3 2016-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file `fun-hard` is gone and the link count shown for `fun` is reduced from four to three, as indicated in the second field of the directory listing. Next, we'll delete the file `fun`, and just for enjoyment, we'll include the “-i” option to show what that does:

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file `fun'?
```

Enter “y” at the prompt and the file is deleted. But let's look at the output of `ls` now. Noticed what happened to `fun-sym`? Since it's a symbolic link pointing to a now-nonexistent file, the link is *broken*:

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2016-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir2
lrwxrwxrwx 1 me   me      3 2016-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure `ls` to display broken links. On a Fedora box, broken links are displayed in blinking red text! The presence of a broken link is not in and of itself dangerous, but it is rather messy. If we try to use a broken link we will see this:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir1
drwxrwxr-x 2 me   me   4096 2016-01-15 15:17 dir2
```

One thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. `rm` is an exception. When you delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this, we will return to our home directory and use rm with the recursive option (-r) to delete playground and all of its contents, including its subdirectories:

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

Creating Symlinks With The GUI

The file managers in both GNOME and KDE provide an easy and automatic method of creating symbolic links. With GNOME, holding the Ctrl+Shift keys while dragging a file will create a link rather than copying (or moving) the file. In KDE, a small menu appears whenever a file is dropped, offering a choice of copying, moving, or linking the file.

Summing Up

We've covered a lot of ground here and it will take a while to fully sink in. Perform the playground exercise over and over until it makes sense. It is important to get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise by adding more files and directories, using wildcards to specify files for various operations. The concept of links is a little confusing at first, but take the time to learn how they work. They can be a real lifesaver.

Further Reading

- A discussion of symbolic links: http://en.wikipedia.org/wiki/Symbolic_link