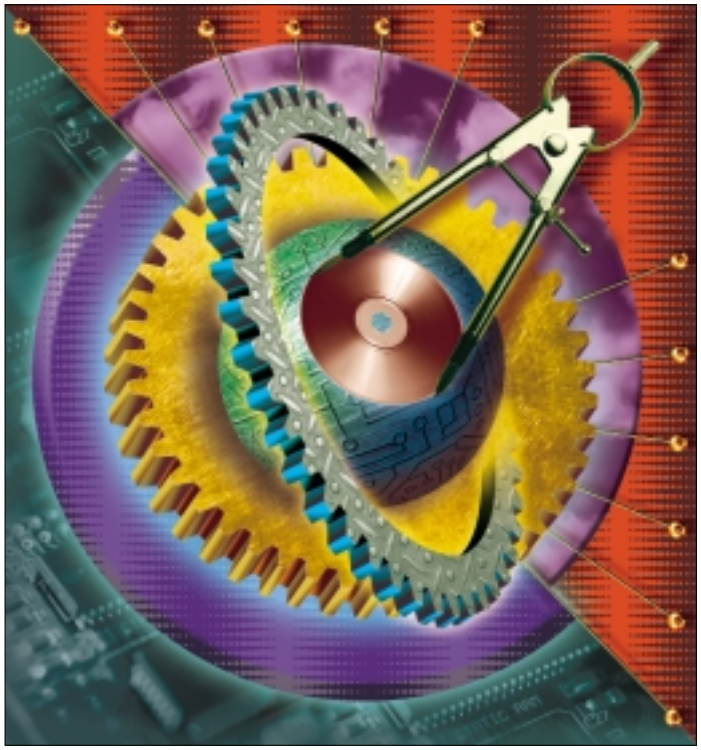
*by Peter Collinson,* **Hillside Systems**



BEK SHAKIROV

# Links

Each UNIX manual page is split into several sections, the first being "NAME" that gives the name of the command followed by a one line description of what the command does. At some point in the early '80s, someone at Berkeley had the bright idea of taking these descriptions, making a searchable database and providing a command to search that database. The `apropos` command was born.

To activate this feature, you first need to use the `catman` command on your machine to generate the database. Then, assuming you can type `apropos` correctly (or at least you know that `man -k` does the same thing), you have a command that can be used to take a keyword and find appropriate manual pages.

Of course, there's an old adage in computing: garbage in, garbage out. The usability of the `apropos` command depends on the text present in the NAME line. I remember one staff member at the university where I worked complaining that they could not delete files on UNIX. They'd tried giving "delete" as the argument to the `apropos` command and nothing useful was printed.

For many years, the `rm` manual page said "rm–unlink files" in the NAME section. Users are unlikely to think that unlink is a satisfactory synonym for "delete." On the Solaris and BSD derived systems I am running, the `rm` command says: "remove directory entries" in the NAME section. This is completely accurate, if somewhat enigmatic. In fact, it's arguable that the manual page for Corel Linux is strictly incorrect because it says "remove files." Perhaps "remove files, usually," would be nearer the actual action of the command.

The reason that the manual page for the `rm` command doesn't say "delete files" and perhaps shouldn't say "remove files" is related to the way that the UNIX file system works. If you've been reading my articles during these opening months of the 21st century, then you would have learned that your disks on a UNIX system are split into several partitions (or slices). Each partition contains a file system (apart from those used for paging or swapping). UNIX joins each of these file systems at runtime into one large hierarchical file store presenting you, the user, with a tree of files.

Each file in each file system consists of some number of disk blocks and the system knows where the blocks for a specific file are stored by inspecting its *inode* (or index node). The inode contains other metadata about the file, such as its ownership, its access permissions, and some time-stamps. However, unlike other operating systems, where the name of the file is retained with the file metadata, the inode doesn't store the name of the file. Names are stored in directories.

The sole task of directories is to store filenames and the associated inode numbers of the files to which each name is attached. Each directory entry is simply a name and an inode number. In the early UNIX systems, a directory entry was a 16-byte fixed length record, two bytes were used to store the inode num-

ber and the remaining 14 held the name that the user had assigned to the file. The UNIX fast file system created by Kirk McKusick changed this structure into a variable length record to permit infinite file names (where infinity is set to 255 bytes). As file systems have become larger, the number of bytes assigned to hold inode numbers have increased too, in turn raising the total number of files that can be held on any particular file system partition. Solaris allows file systems that support 64 bit inode numbers, although my system seems to be set to use 32 bits.

When a process on the system opens a file, it will supply a pathname to the kernel, and the first task of the kernel is to follow the path to find the inode for the file that is to be used. For example, if the user has typed `/bin/ls` into their shell, the shell will eventually ask the kernel to load the command and run it. The kernel starts at the root of the file system, looking for inode No. 2, which is the root directory. The contents of this directory are read, while searching for the name `bin`. On my system, the root directory says the bin directory is inode 16, so this inode is read and the disk blocks that it contains are pulled from the disk to permit the kernel to search for the `ls` string. Eventually, the `ls` entry is found, and its inode determined (89948 on my system). This is the target file and will be loaded into memory and executed.

## Links

As we have seen, the job of the directory is to hold a name and an inode number for each file the directory contains. There are three constraints on the name: first, it must be shorter than 256 characters; second, the name must be unique in that directory. Finally, there's a tiny constraint on the characters that make up the name: the name cannot contain a null byte. It's perfectly possible to create files with names that contain characters that are difficult to type directly into the shell, such as Control-C, or which the shell and other UNIX tools find difficult to deal with, such as a space or hyphen.

It's also very possible to create two or more directory entries that point to the same inode number. We may have several reasons for wishing to do this. First, we may want to have a command that's called by a set of names, and acts differently depending on how it's called. The `vi` editor is an example of this need. The `vi` command can also be called by `ex`, `edit`, `view` and `vedit` but behaves differently for each name. Second, it may be convenient to place a file in several different locations on the machine. Perhaps the file is accessed by a compiled program and is forced to be placed in a known location on the file system, but we would prefer to maintain it from elsewhere. The list of uses for having the ability to create two directory entries for the same file is quite endless.

The command that creates a new directory entry for an extant file is `ln` (*link*). When we say:

```
$ ln oldfile newlink
```

the pathnames are passed into the kernel which looks up `oldfile` to find its inode number and then creates a new directory entry, `newlink`, with the same inode number as `oldfile`. You can see this happen by giving the `-i` option to the `ls` command:

```
$ ls -i
    18720 oldfile
$ ln oldfile newlink
$ ls -i
    18720 newlink      18720 oldfile
```

We have made a *link* to `oldfile` called `newlink`. Notice that the file itself is not touched by this operation, all that has happened is that we have created a new directory entry with an alternate name. All the metadata for the new file will be the same as that of the old because the link actually points at the same inode, and it's the inode that contains the file information.

We're not constrained to create the link in the same directory, a link can be created anywhere. The only limitation is that the link must be on the same disk partition, because inode numbers are local to the file system that lives in that partition. We'll see later how things have evolved to get around this restriction.

Once we can create links to files, we have a problem with what we do when we apply the "remove file" command to one of the files. For example, we need to know what happens when I use the `rm` command on `oldfile` having created the new link above. The command calls the system call to remove the named directory entry but leaves the contents of the file and its inode intact because it is referenced by the other directory entry. The kernel will only delete the file contents and its associated inode when it removes the last directory entry that references this information.

The kernel knows how many directory entries point to a specific inode by inspecting the value of a reference count that's stored in the inode. In fact, when we create a link, the kernel increments a reference count in the inode saying that the file now has more than one directory entries that point to it. This reference count appears in the `ls` listing when we give it the `-l` (*ell* not one) option:

```
$ ls -l
total 76
-rw-rw-r--   2 pc ............... newlink
-rw-rw-r--   2 pc ............... oldfile
```

I've deleted some of the listing for ease of printing. The No. "2" here is the reference count on the inode. Now when I remove `oldfile` using the `rm` command, the directory entry for `oldfile` will be expunged, and the reference count on the inode will be decremented. However, the inode and its contents will not be deleted because the new reference count is not zero. An entry in the file system still points to the inode, and its contents are retained until I delete `newlink`.

The behavior was the basis of the original `mv` command. When you typed

```
$ mv oldfile newfile
```

the program checked that the source and destination files were

on the same partition, and, if so, created a link to `oldfile` called `newfile`. When this was done, `oldfile` was deleted. The effect is to rename the file. If the files were on different partitions, then `mv` had to copy the file, creating a new inode and copying the file contents to the destination partition before deleting the original file.

Many utilities (especially editors) use this type of renaming technique when writing back the edited file. When creating the new file, the utility will make a file with a temporary name in the same directory as the file to be replaced. The new data in the file is written to the disk, and the temporary file closed. To move the data into place, the utility issues a `link` system call that renames the temporary filename to the destination file. If the destination file exists, the system will delete it (or unlink it).

The point here is that it's rare for a `link` system call to fail, while many things can go wrong while writing data to a disk. The primary cause of failure is running out of disk space, but if the editor can get the file safely onto the disk and the temporary file closed, then it's fairly safe to delete the original unedited contents because a copy exists on the media.

**The UNIX file system was designed to be a directed graph, so theoretically there should only be one path through the tree to any particular file.**

It's useful to know this is going on, because when you edit a file that has more than one link, you will destroy the linkage information. Your intention is to update the file, but you will only update one instance of it. You need to go around all the other links ensuring that they point to the right inode.

Kirk McKusick implemented a rename system call in the kernel as part of the change to the Berkeley Fast file system, he's always said that it was the hardest thing he had to do. One reason for putting the code in the kernel was to support remote file access via NFS more easily. Another was that the effect of the original `mv` command was somewhat uncertain if the system crashed during its operation, or worse, some other process started to mess with the file that was being renamed.

## Links and Directories

In these articles, I often say that a directory is just a file whose format the kernel understands, but it is a fairly special file. Losing a directory near the top of a file system can be catastrophic and the system works hard to stop such accidents from happening. Processes are prohibited from changing a directory by any means other than the standard set of system calls. Actually, anyone can read a directory, try:

```
$ od -c .
```

to see the full gory details of the format.

When a directory is created, it is automatically furnished with the "." (the current directory) and ".." (the parent directory) entries. If you use `ls -ld` on a newly created directory, you'll see that the inode reference count is set to two reflecting the presence of these two links. Essentially, the `mkdir` system call that creates a new directory considers that the new directory is linked into the old. Since we are adding a link, we update the link reference count on the inode of the parent directory. The reference count now tells you the number of directories that a specific directory contains. To test this, just make a new directory in the directory you have created and watch the reference count increase by one.

The reference count on the directory doesn't seem to be of much use apart from consistency. It's certainly true that directories cannot be removed unless the reference count in their inode is set to two. But in order to delete a directory, we still have to search it to ensure that it doesn't contain any regular files. So, at best, the check for the reference count being two only saves speed when we have finger trouble and attempt to delete a directory that shouldn't be deleted anyway.

The UNIX file system was designed to be a directed graph, so that theoretically there should only be one path through the tree to any particular file. The file system is not supposed to have connections at the directory level where one directory points to another, which in turn contains a directory that is the first directory. Doing this would give us possibly infinite loops when we are searching for a particular file. For this reason, it's not possible to link directories in the file system. Notice that prohibiting directory links means there is a single path from the root to a particular file entry in a directory. Also, there is a single path back up the tree from the directory entry to the root. This means that pathnames that involve ".." are deterministic and the fact that there's a unique route back up the tree is an important feature of the file system.

## Symbolic Links

We've seen that since links are based on inode values, they are limited to linking within the current file partition. Also, we cannot use links to link directories. There are often situations where these limitations really get in the way of what is needed. Symbolic links are another innovation from Berkeley that get around the limitations and, of course, do cause a few new problems of their own.

The term "Symbolic link" is often abbreviated to *symlink*. While I am talking about names, you'll find that the original UNIX inode based links are often called *hard links* to make a firm distinction between the two types of link.

A symlink is simply a file that contains the name of another file. When the symlink is opened or accessed in normal circumstances, then the system notices that you are dealing with a symlink and will automatically open or access the file whose name is found in the symlink's contents. To create a symlink, all we do is to add the `-s` flag to the `ln` command and proceed as before:

```
$ ln -s oldfile newlink
$ ls -l
total 76
lrwxrwxrwx   1 pc ..... newlink -> oldfile
-rw-rw-r--   1 pc ..... oldfile
```

Again I have truncated the output for printing. The `ls` command notices that `newlink` is a symlink and prints the destination file name. Actually, you can make `ls` "resolve the link" by giving it a capital "L":

```
$ ls -lL
total 76
-rw-rw-r--   1 pc  ..... newlink
-rw-rw-r--   1 pc  ..... oldfile
```

Thanks to David Robinson from `EBay.Sun.COM` for this useful tip.

The name that's stored in a symlink follows the normal path construction rules, and in the example above it's pointing to a file in the current directory. There is no reason why it cannot contain any path that can be resolved on the machine. I often have symlinks that point from one machine to another, hiding the complexity of network mount points. I also use symlinks for convenience. For example, I've kept the personal source tree on `/s` for eons, its actual location has wandered around my filestore depending on where the free space was to be found. However, I am in the habit of typing `/s` to get to a file in this tree, and by making `/s` into a symlink I can always do this.

The target file doesn't have to exist when you are setting up a symlink. I can type:

```
$ ln -s turnip bad
$ ls -l
lrwxrwxrwx   1 pc ..... bad -> turnip
lrwxrwxrwx   1 pc ..... newlink -> oldfile
-rw-rw-r--   1 pc ..... oldfile
```

I've created a new symlink called `bad`, but the target file to which it points doesn't exist. The rationale here is that it would not be sensible to check on the existence of the target, when I can always create a targetless symlink by deleting the target file after I've created the link. In fact, one of the problems with symlinks is that it's too easy to have a whole nest of links that point nowhere.

However, if you delete a target and re-instate it with a new copy, any symlinks that pointed to the file are still in place and will point to the original file name. I tend to have a bunch of symlinks in my private `bin` directory that point to system files. For example, I create symlinks to the small number of commands that I use on, say, `/usr/sbin` to avoid having to put the directory in my search path. I use this technique to pick up the Berkeley versions of `ps` and `df` from `/usr/ucb` without having to buy into all the Berkeley versions of system commands. Using a symlink for these commands means that the binaries can be updated when I install a new system, or a system patch, and I know that the command I am running is up-to-date.

Symlinks behave like regular files–to delete a symlink you just unlink it from the directory in which it lives using the standard `rm` command. You may think that their ownership and permissions behave oddly. When you create a symlink, it's apparently owned by you and will have a set of file access permissions that depends on the system you are using. Solaris sets `rwxrwxrwx` into the symlink. However, neither of these settings affects the operation of the link, the permissions and ownerships that matter are the ones attached to the target file.

Symlinks break the rule that the filestore should be a directed graph. It's perfectly possible to point a symlink at a directory and create parallel paths through the filestore. When you change into the directory, the shell makes a jump from one place in the tree to another. As a consequence, moving back "up" with ".." can give you a surprise–you are not back where you started, but in some completely different directory that's the "real" parent of the target directory.

Some shells (notably `ksh` and `bash`) have recognized that users prefer ".." to work "correctly." When you descend a tree following a certain route, then going back up should reverse that route. The shells store the path you have taken through the tree, and use the information to make `cd ..` step back up the path rather than read the ".." from the file system and follow it back up the "real" parent directory. It's probable that this behavior is desirable and convenient, but can be confusing. You are being lied to about exactly where you are in the regular file tree. You can easily become "temporarily uncertain of position" as pilots who are lost would say to make it seem better.

In these cases, you'll find that typing `/bin/pwd` can help. The `pwd` command is usually a shell built in and will happily lie to you about where you are, based on what it thinks you want to know. However, the command on `/bin` isn't party to the deceit and will backtrack up the tree to find your "real" position.

It's also very easy to set up loops in the filesystem where one symlink points to another, which points back to the first. For this reason, the kernel does some checking to ensure that it doesn't get stuck in an endless loop when accessing the file at which a symlink points.

However, as we've seen, symlinks allow us to create links that don't suffer from the problems of hard links. Symlinks finally allowed us to create file system hierarchies that are independent of the location of files in the file system that's glued to the disk surface. However, I should end with a note of caution. It is very easy to become deeply confused about exactly where target files are on your system. As with many things, KISS.

---

***Peter Collinson*** *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever ... He writes, teaches, consults and programs using Solaris running on an UltraSPARC/10. Email:* `pc@cpg.com`.