Franz Pfluegl, Fotolia

Data management in the Shell

WELL ORGANIZED

Do some serious spring cleaning and reorganize your data. The right commands can help you to keep on top of your file and directory management. **BY HEIKE JURZIK**

Where graphical file managers like Konqueror and Nautilus use mouse clicks and convenient features such as drag and drop, you could use some simple commands in the shell. This article introduces the *mkdir*, *rmdir*, *cd*, *touch*, *cp*, *mv*, and *rm* commands, and demonstrates highly efficient data management at the console.

Creating and Deleting directories

The *mkdir* ("make directory") command lets you create a new folder. The following command:

```
mkdir folder1
```

creates a directory named *folder1* in your current directory. The access privileges for the new directory are defined by the umask (see the "Umask" box on page 88). To assign different permissions, you can use the *-m* flag, with an octal number to define the permissions you want (Listing 1).

The command also understands relative and absolute path values. For example, to create a folder below the *music* directory, you don't actually need to change to the directory. Instead, you just specify the path:

```
mkdir music/Metallica
```

If the superordinate directory does not exist, *mkdir* will protest:

```
mkdir: cannot create directory ➥
 `music/Metallica': No such ➥
  file or directory
```

In this case, you need to set the *-p* option to create hierarchies of folders at a single step. Instead of entering the following list of commands:

```
mkdir music
mkdir music/Metallica
mkdir music/Metallica/Load
```

you could achieve the same effect with the *-p* parameter and a single command:

```
mkdir -p music/Metallica/Load
```

Commands

- Mkdir – Lets you create directories
- rmdir – Lets you remove directories
- Cd – Helps you navigate between folders
- cp – Copies data
- mv – Moves things
- Rm – Gives users an elegant approach to getting rid of files and directories

Listing 1: -m flag

```
01 $ mkdir folder1
02 $ mkdir -m 777 folder2
03 $ ls -l
04 drwxr-xr-x  2 huhn huhn 4096
    2006-12-28 14:07 folder1/
05 drwxrwxrwx  2 huhn huhn 4096
    2006-12-28 14:08 folder2/
```

The command for deleting a directory is *rmdir* ("remove directory"). If the folder is not completely empty, *rmdir* will refuse to cooperate:

```
$ rmdir folder1
rmdir: "folder1/": ➥
 Directory not empty
```

In this case, you can either remove the folder content first, or just call the *rm* command (see the "Clean Slate" section), which has an option for enforced deleting. *rmdir* also supports the *-p* parameter, and assuming that all the folders are empty, the following command:

```
rmdir -p music/Metallica/Load
```

will remove all three directories from your disk in one fell swoop.

The Right Touch

The *touch* command is often used to create new, empty files. The command:

Umask

The umask defines the permissions the filesystem assigns to new files and directories. You can type *umask* at the prompt to output the current value for this variable. The output will be a four-figure octal number that defines the permissions to be denied.

In our example, *0022* means that text files, which are typically created in *0666* mode (read and write access for all), are assigned *0644* (*0*666 minus 022) instead (that is, *-rw-r--r--*). For directories, the default permissions are *0777* (all permissions for all); with a umask of *0022*, directories are set to *0755* (*drwxr-xr-x*).

You can also use the *umask* command to modify the mask itself. To make your changes permanent, just enter the command in your Bash configuration file *.bashrc* below your home directory.

Figure 1: cd - takes you to the last directory you visited, and back again.

```
touch cluck
```

creates an empty file called *cluck* in the current directory, assuming the file does not already exist. The command also evaluates the umask to automatically set permissions.

If the file exists, *touch* modifies the **timestamp** for the file, and sets the last access and last changed times for the file to the current time. This makes a lot of sense in combination with make, for example. However, this command only works if one or multiple source files have changed.

To talk make into running, even if the sources have not changed, you can run touch against the source code files to modify the timestamps, as in touch *.tex, for example.

Clever Navigation

The *cd* ("change directory") command lets users navigate folders. You can specify either an absolute path, or a relative path:

```
cd /var/log
cd ../../var/log
```

The command has a couple of practical shortcuts. Typing *cd* without any other parameters takes you to your home directory.

cd evaluates the *HOME* **environmental variable** to discover the path to the home directory:

```
$ echo $HOME
/home/huhn
```

Bash also uses the *OLDPWD* environmental variable to store the name of the last folder you visited, and *cd $OLDPWD* takes you back to this folder.

Typing *cd $OLDPWD* a second time beams you back to your previous directory; the shell only stores these two directories, so that you just jump back and forth each time you give the *cd $OLDPWD* command. You can save some typing by replacing *$OLDPWD* with a minus sign (Figure 1).

Copying Files and Directories

The *cp* ("copy") program duplicates files. To use the command, you need to specify the source and the target:

```
cp file1 file2
```

The copy this command creates has the current timestamp, and belongs to the current user by default. Again, permissions are controlled by umask. If you need to keep as many properties of the original file as possible, you can set the

GLOSSARY

Timestamp: Unix filesystems manage a number of time entries for a file, such as the last access or last modified times. It is particularly important for backups, for example, to keep the original timestamp (e.g., of the last modification) to avoid working on the wrong file.

Environmental variable: The Shell gives each user space in which to store specific information for access by programs. Environmental variables comprise a name and a value.

Symbolic link: A pointer to another file that is treated just like this file by application programs. If you delete the file to which the symlink points, the link becomes orphaned. Symlinks are created by the *ln -s* command.

-p flag to keep permissions and the source timestamp.

You can also specify a directory as the target. Although you can copy multiple files to a directory, the program will not copy directories:

```
$ cp folder1 folder2
cp: omitting directory➥
 "folder1"
```

To copy a directory, you need to tell *cp* to use recursion, by setting the *-r* option. However, this is not all the copy artist can do; if you tell the command to copy a **symlink** instead of a "normal" file, cp will copy the file to which the link points. To create a new link instead of doing this, you need to set the -d flag.

Be careful if you specify a file of the same name as the target; *cp* will just overwrite the existing file. You can add a safety net by specifying the *-i* option:

```
$ cp -i file1 folder/file1
cp: overwrite "folder/file1"?
```

Alternatively, you can talk *cp* into creating a backup by stipulating the *-b* option:

```
cp -b file1 folder/file1
```

The backup copy has a tilde (" ~ ") at the end of its filename.

Let's move!

The *mv* ("move") tool can either move or rename files. Again, you need to pass a source and a target to the command. The source can be a file or a directory, and the target can be a directory, or a file or directory name. To move a file called *file1* from the current directory to an existing folder called *directory1*, simply do the following:

```
mv file1 directory1
```

Listing 2: rm

```
01 $ ls -l
02 -r--r--r--  1 huhn huhn
       0 2006-12-28 16:44 file
03 ...
04 $ rm file
05 rm: remove (write-protected)
    file "file"?
```

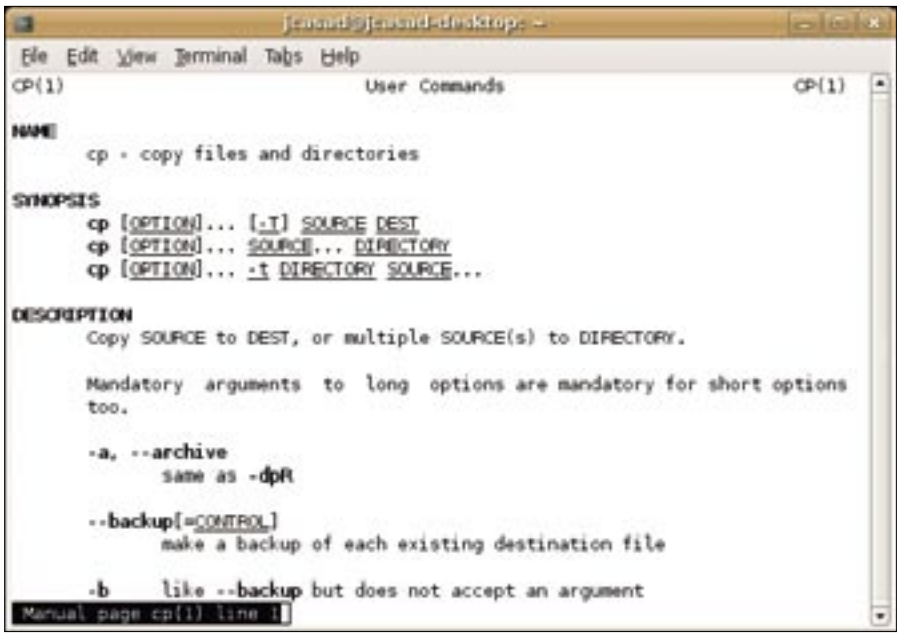

Figure 2: If you forget the command-line options, use the man command to view the manpage.

If the target folder does not exist, this command will change the file's name to *directory1* – in other words, you just need to specify the new name to rename the source.

Just like *cp*, *mv* supports the *-i* and *-b* options to protect data against inadvertent deletion by prompting or creating a backup.

Clean Slate

The *rm file* command typically deletes the specified file without so much as a "by your leave". *rm* ("remove") doesn't hang about, and doesn't prompt you to see if you are really sure. The only exception to this is if you have a read-only file; in this case, *rm* prompts you to confirm (see Listing 2).

However, this is not a recommended approach. Instead, you might prefer to use the *-i* option, which switches the command to interactive mode (just like it does with *cp* and *mv*):

```
$ rm -i file
rm: remove regular file➥
 "file"?
```

If you have a large number of write-protected files in a folder, and don't like the idea of having to confirm removal of each file, you can use the *-f* ("force") parameter to tell *rm* not to ask.

For more thorough deletion, *rm* also has the *-r* option. If you set it, *rm* will happily remove subdirectories – along with their content – recursively up to the top of the path. Where *rmdir* hesitates to remove a folder if it is not completely empty, *rm -r* just blows it away. If you are interested in seeing what is going on behind the scenes, and where *rm* went, just set the *-v* flag (Figure 2).

Files that start with a non-standard characters, such as a minus sign, are problematic; *rm* (and other commands) will refuse to touch them. The reason for this is that the shell interprets the first character after the minus sign as a parameter, and will thus not be able to find the intended target.

A trick helps you resolve the situation. Type *rm ./-file* to delete *-file*, rather than interpreting the target as a parameter. As an alternative, you can insert two dashes in front of the filename (*rm -- -file*); this tells the command that whatever follows the dashes is not a parameter, but rather an argument (that is the target for the current operation). ■

THE AUTHOR

Heike Jurzik studied German, Computer Science and English at the University of Cologne, Germany. She discovered Linux in 1996 and has been fascinated with the scope of the Linux command line ever since. In her leisure time you might find Heike hanging out at Irish folk sessions or visiting Ireland.