# A Very Valuable Find

By Jerry Peek

Unless you're superbly organized or have a great memory, chances are you "misplace" files from time to time. *Find* answers the question, "Now, where did I put that?"

Of the many file management utilities in Linux, *find* is worth understanding in depth because it does so much more that just locate files. *Find* lets you write decision-making logic, specify filters, and run commands on the contents of entire directory trees. Unfortunately, *find* is not documented well, and its syntax can appear complex. But, as we'll see, *find* is actually simple to use, elegant, and a great Linux power tool.

## In Search Of...

More on *find* features soon — first, let's look at a simple example and some *find* fundamentals. (If you've never used *find* or want to see more examples, see "Finding Stuff" in the April 2001 issue, or online at http://www.linux-mag.com/2001-04/newbies_01.html.)

The following command looks for all files named `foo` in the current directory (through the shortcut `.`) and all of its subdirectories:

```
$ find . -name foo -print
./junk/foo
./proj/2002/06/foo
```

The first argument to *find* is the pathname of the directory where *find* starts its search. Here we're using `.` (dot), the relative path for the current directory. *find* searches `.` and all subdirectories within it. For each directory entry (including files, symbolic links, sockets, and "hidden" entries whose names start with a dot, like *.mozilla*) *find* performs a test, which is expressed with *operators*.

Some operators require arguments; others stand alone. *find* tests each directory entry against the operators, from first to last (in command line order). If any of the operators don't match (if the result isn't "true"), *find* skips the directory entry and goes on to the next.

In this case, there are two operators. `-name foo` is true if the file's name is exactly *foo*. If true, *find* will evaluate the next operator, `-print`, to see if it's "true." Actually, `-print` is *always* "true," and always prints the pathname of the current directory entry. (There's more about `-print` in the sidebar "To -print or not to -print.") As you can see, the example *find* command discovered two entries named *foo* in subdirectories of `.` and printed their names.

## Logic and Order

If you're familiar with any programming or scripting language, you're probably familiar with *short-circuit evaluation* of logical expressions. In logic, an expression "A and B" is true *only* if *both* A and B are true. If expression A isn't true, there's no way the whole expression can be true, so there's no need to test B. Hence, you can "short-circuit" the process.

You can think of *find*'s operators as logical expressions that are joined by invisible "and"s (and, as we'll see later, by visible "or"s) into logical expressions. And, like a smart programming language, *find* short-circuits as soon as it's obvious that an expression isn't true.

In the previous example, you can think of `-name foo` as the "A" and `-print` as the "B", with an implicit "and" between them. If `-name foo` isn't true, then *find* will short-circuit the evaluation and not bother to test `-print`. In other words, if the file isn't named *foo*, then *find* won't print its name.

Combining operators to express what you want to find is the key to unlocking *find*'s power. And, because *find* short-circuits, the *order* of operators is very important. For instance, what would you expect to happen if you put `-print` first (before `-name foo`)? Every pathname would be printed. Remember, `-print` always evaluates to true. If `-print` is first in a logical expression, it always prints a pathname as a side effect of being evaluated. The logical expression may short-circuit later, but that doesn't change the output.

You can put any number of operators on the command line. For instance, to find only *files* named *foo* — not directories, and so on — add `-type f` (before `-print`, of course). Then, the `-print` operator will be evaluated only if both the `-name` and `-type` operators are true.

## Efficient Ordering

Choosing the right order can also make *find* more efficient. That's important if *find* is searching a lot of files. For example, testing a file's name is easy: *find* already knows all the filenames in a directory. Using `-exec`, which executes an entirely new Linux process, is slower. So, as you order the operators for a deep search, try to think about how *find* might implement it.

For instance, the next example looks for every file in the */prj/reports* directory whose name ends with *.bak* that also contains the word `DRAFT`:

```
$ find /prj/reports -type f -name '*.bak' \
      -exec grep -s DRAFT {} \; -print
```

(Backslashes and quotes are often needed when using *find*. For more information, see the sidebar "Special Characters: Shells vs. *find*.") Let's focus on the arguments that *find* gets from the shell. That rather long operator, `-exec grep -s DRAFT {} ;`, runs the Linux utility `grep -s` to search the current file, which *find* abbreviates as `{}`. The *grep* option `-s` ("silent") tells *grep* not to output matching lines, but simply return an exit status of 0 if it finds the word `DRAFT`. An `-exec` operator is "true" if the program it runs returns a 0 exit status. So, if *grep* finds a match, it returns 0, the `-exec` succeeds, and then *find* evaluates `-print`, which prints the filename.

If we had put the `-type` and `-name` operators *after* `-exec`, then *find* would have run *grep* on *every* entry — even on subdirectories and on files with the wrong names — which would have wasted a lot of time!

## Grouping and Multiple Expressions

As mentioned above, `find` puts an implicit "and" operator between all operators. (You can write *find . -name -foo -a -type f -a -print*, but that's not required.) *Find* also has "or": it's the `-o` operator. "Or" — which you always have to write explicitly — lets a test succeed if *any* of a chain of or-joined operators is true.

Here's an example. If your username is *ed* and your group is *staff*, you could find all directories where you have write permission by user and group with the command shown in *Figure One*.

The parentheses are *find* grouping operators. (The backslashes are required to keep the shell from interpreting the parentheses.) Why do we need them? You have to tell *find* which comparisons to make. "And" has higher precedence than "or" — so, *without* the parentheses, the expression would be true either if both the `-type d` and `-user ed` tests are true, or if both the `-group staff` and `-perm -220` tests are true. We don't want that!

Grouping with parentheses makes the test true if `-type d` is true *and* if *either* `-user ed` *or* `-group staff` is true. Of course, the `-perm -220` test (which checks for the "write" permission, bit 2, for user and group) also has to succeed before `-ls` can happen.

You can use this "either-or" behavior in another, less obvious way: to make a long *find* command line with multiple expressions, only one of which will be used for a particular directory entry. *Figure Two* shows an example that an administrator might run nightly, from a shell script, to clean up a directory tree.

What's happening there? Let's take it line by line:

- The first line tells *find* to search the */prj* tree. The `-type f` test is applied to all entries. If the entry isn't a file, then short-circuit evaluation skips the rest of the test. (Remember: because there's no operator after `-type f`, that's an implicit "and".) The second half of the "and" is the entire expression after the first grouping parenthesis.

- The next three lines are three subexpressions with "or" (`-o`) operators between them. An "aerial" view looks like this: *((first-expr) -o (second-expr) -o (third-expr))*. The outer parentheses surround all expressions because the expressions are also joined with "or"s, any one can be true.

- *find* evaluates each subexpression, starting with *first-expr*. If it's true — if the filename ends with a tilde (`~`) *and* it was modified more than 7 days ago — then the `rm -f` command is executed to remove the file. Assuming that *rm* succeeds (returns a status of 0, which the `-f` option makes very likely), then `-exec` is true, the whole subexpression is true, and *find* stops evaluation for this file.

- If "first-expr" wasn't true, *find* tries *second-expr*. Is the file named *core* and was it last accessed more than 30 days ago? If so, remove it and "second-expr" is true.

- If neither of the first two subexpressions succeeds, the third can. This one looks for filenames ending in *.bak* (backup files, perhaps) that are *not* read-only for all users. (The `!` operator reverses the value of the expression after it: if `-perm 444` isn't true, `! -perm 444` *is* true.) If the permissions aren't 444, *find* executes *chmod* to make it so.

The advantage of these long expressions is that *find* traverses the directory tree *only once* — as opposed to your other choice, running a series of *find* commands that have to separately traverse the whole directory tree. The disadvantage

***See Power,*** *pg. 61*

**FIGURE ONE:** Using an "or" operator

```
$ find / -type d \( -user ed -o -group staff \) -perm -220 -ls
927441 4 rwxrwxrwx 39 ed staff 4096 Jun 25 10:51 /home/ed/tmp
134001 4 rwxrwx--- 2 jan staff 4096 Jun 23 18:23 /prj/adir
...
```

**FIGURE TWO:** Testing three things with one *find* command

```
$ find /prj -type f \
\( \
  \( -name '*~' -mtime +7 -exec rm -f {} \; \) -o \
  \( -name core -atime +30 -exec rm -f {} \; \) -o \
  \( -name '*.bak' ! -perm 444 -exec chmod 444 {} \; \)
\)
```

***Power,*** *from pg. 45*

comes if you want more than one subexpression to be executed for a particular file: joining them with "or" means that only one succeeds.

## Phenomenal *finds*

Now that you've seen the most important parts of how *find* works, here are some examples.

**1.** Find uncompressed files (whose name doesn't end with *.gz* or *.bz2*) with more than 10,000 characters, show a long listing of the file information, then ask if you want to compress the file with *gzip*:

```
$ find archive -type f \
  ! -name '*.gz' ! -name '*.bz2' \
  -size +10000c -ls -ok gzip {} \;
```

In this example, we're using two "action" operators. The first (`-ls`) always succeeds, so the second (`-ok`) always happens after `-ls` shows the file listing.

**2.** Same as above, but always try to compress files. If *gzip* fails (doesn't return 0 status), start a shell to (optionally) fix the problem. When you type *exit* to end the shell, *find* will go on to the next entry:

```
$ find archive -type f \
  ! -name '*.gz' ! -name '*.bz2' \
  -size +10000c \
  \( -exec gzip {} \; -o -exec bash \; \)
```

This depends on `-o`: if the first `-exec` fails (because *gzip* failed), *find* will try to make the parenthesized expression "true" by evaluating the second `-exec` to run the *bash* shell.

**3.** Duplicate the subdirectory tree under the current directory (don't copy files, just make empty subdirectories):

```
$ cd /prj/daily/20020626
$ find . -type d ! -name . \
  -exec mkdir /prj/daily/20020627/{} \;
```

This example depends on *find* visiting a directory before its subdirectories, which it always does (unless you use the `-depth` operator). This actually executes commands like `mkdir /prj/daily/20020627/./adir`, but you can ignore the `./` (which stands for the current directory along the pathname); the result is `mkdir /prj/daily/20020627/ adir`). The "`! -name .`" prevents trying to re-create the destination directory with `mkdir /prj/daily/20020627/.`. Note that this last example doesn't work on non-GNU versions of *find* that expect `{}` to stand alone.

If you'll be using *find* on more than one system, be sure they all support the operators you want to use. But all versions work the way you've seen here: treating their operators logically, with short-circuit evaluation. If you keep that in mind, you'll know how to use the real power of *find*.

Next month we'll see what's really behind pathnames and the current directory, and also see lots of tips for getting what you want from the filesystem.

*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 at jpeek@jpeek.com.*

### TO -PRINT OR NOT TO -PRINT

Your Linux version of *find* probably doesn't require `-print`. If you don't use any "action" operator like `-ok`, `-exec`, or `-print`, the GNU *find* will add `-print` automatically. This might irritate people who love the beauty of *find* expressions, but it solves an old problem: if you forgot `-print`, *find* would seem not to have "found" anything.

### SPECIAL CHARACTERS: Shells vs. Find

The *find* command

```
$ find archive -type f \
  ! -name '*.gz' ! -name '*.bz2' \
  \( -exec gzip {} \; -o -exec bas
```

shows an important thing about *find*: the backslash (\) characters. Actually, *find* never sees those backslashes. They're there for the shell, which reads the command line and passes (most of) its arguments on to *find*. You'll use lots of backslashes with find because *find* command lines tend to be long and because a lot of *find*'s special characters (like `*` and `;`) are also special to the shell.

The first backslash, at the end of the first line, tells the shell to keep reading arguments on the following line; it's not passed on to *find*. On the third line, backslashes before the parentheses tell the shell not to treat them as subshell operators; the parentheses *are* passed on to *find* as grouping operators. In the same way, backslashed semicolons (`;`) aren't interpreted as shell command separators; the backslashes are removed and the semicolons are passed to *find*.

(We also could have used backslashes before the stars (`*`) in the filename patterns and written `\*.gz` instead of `'*.gz'`. Both have the same effect: telling the shell not to interpret the star (not to expand `*.gz` at the time that the command line is interpreted) but to pass it on to *find* (so find will interpret the `*.gz` as it checks the name of each directory entry.)

If you're new to the shell and the command line, all the backslashes may seem intimidating. But they're actually just another way to extend the syntax and give the Linux command line even more power. Learn how Unix shells work — they're one of the most important power tools.