POWER TOOLS

What's GNU, Part Four

By Jerry Peek

A few months ago, "Power Tools" presented the third article in an ongoing series about new features added to long-standing *Linux* and *Unix* utility programs. This month, let's pick up again with the "What's GNU" series and look at the new features that GNU programmers and others have added to the ubiquitous *find*. (If you're not yet familiar with *find*, you can find an introduction to the utility in the September 2002 "Power Tools" column, "A Very Valuable Find," online at http://www.linux-mag.com/2002-09/power_01.html.)

There are lots of versions of *find*. All examples shown here are based on GNU *version* 4.1.20 (which is the latest version of the code included in the *Debian stable* distribution as the magazine goes to press).

Filename Matching

Older versions of *find* have only one way to check the name of an entry: the `-name` test, where the argument to `-name` is a case-sensitive filename or a shell wildcard pattern like `*.{c,h}`.

While a shell wildcard is simpler to use than a *grep*-like regular expression, it limits the matching that `-name` can do. For instance, matching a filename that contains all uppercase characters is tough to accomplish with shell wildcards. On the other hand, matching all uppercase characters is simple with a regular expression (as you'll soon see with `-regex`).

`-name` has other limitations, too. The string or wildcard pattern after `-name` is compared only to the *name* of the entry currently being scanned, not the entire pathname of the entry. So, for instance, it's easy to know whether the current filename ends in `.c`, but it's a lot harder to know whether that file is in a directory named *src*.

The `-path` test, which was added fairly early to many versions of *find*, performs a shell wildcard-like pattern match against the entire current pathname. So, the test `-path '*src/*.c'` works better: it matches any pathname containing `src`, followed by any number of characters and ending (eventually) with the literal `.c`. That could be the file *./src/foo.c*, but it could also be the file *./src/subdir/bar.c*, *./TeXsrc/foo.c*, or something even messier. The matching of `*` is wide-open — `*` means "zero or more of any character" — so be careful using it if you need a specific pathname match.

The GNU *find* has several other name tests:

- `-iname` matches a case-insensitive shell wildcard pattern.

- `-regex` compares names to a case-sensitive regular expression. Like `-path`, the regular expression is tested against the entire current pathname. To test just the filename of the current entry, ignoring the leading path components, use a regular expression that starts by matching all non-slash characters up to the final slash (`^[^/]*/`). For instance, to match filenames that are all uppercase ASCII letters, try `-regex '^[^/]*/[A-Z]+$'`.

- `-iregex` works like `-regex`, but ignores case.

Another new test is `-lname`. It matches the *target* of a symbolic link (or symlink). (Other name tests, like `-name`, match the name of the symbolic link itself.) The corresponding `-ilname` test does case-insensitive matching of the symlink target.

There are also two other new tests and options for symbolic links: `-follow` dereferences each symlink, finding the file the link points to, and `-xtype` is like the opposite of *–type* for symbolic links. If `-follow` isn't given, `-xtype` checks the file that the symlink points to; otherwise, `-xtype` checks the symlink itself.

Matching Timestamps

Older versions of *find* matched timestamps only in 24-hour

DIVING INTO THE DEPTH OPTION

Because *find* is often used to give filenames to archive programs like *tar*, it's worth understanding `find -depth` and how to combine it with *tar*.

A *tar* archive is a stream of bytes that contains header information for each file (including its name and access permissions) followed by that file's data. The archive is extracted in order from first byte to last.

Let's say that you want to archive an unwritable directory. When you later extract that directory from the archive, its permissions will be set at the time it's extracted.

If you don't use `-depth` to create the archive, the directory will be extracted *before* its contents, meaning that the entries in that directory can't be extracted — unless *root* is extracting the archive — because the directory that contains the files isn't writable at that point.

However, if you use `-depth`, the problem is solved. That's because, when *tar* extracts a file before its directory, it temporarily creates a writable directory to hold the file. Then, later, when *tar* extracts the directory itself, the directory's contents have already been extracted and all *tar* has to do is to set the directory's new (unwritable) permissions.

intervals. For instance, the tests `-mtime -3` and `-mtime -2` are both true for files modified between 72 and 48 hours ago, respectively. Besides being a bit hard to understand at first, the three timestamp tests (`-atime`, `-ctime`, and `-mtime`) are likewise limited to 24-hour granularity. If you needed more accuracy, you'd have to use `-newer` or `!-newer` to match the timestamp of a reference file — often one created by *touch*. (Worse yet, many versions of *find* would silently ignore more than one `-newer` test in the same expression!)

The new `-amin`, `-cmin`, and `-mmin` tests check timestamps using *minutes* as the unit of measure. For example, to find files accessed within the past hour, use `-amin -60`. (It's hard to test last-access times for directories. When *find* searches through a file tree, it accesses all of the directories — updating all directories' last-access timestamps in the very process.)

Another new option, `-daystart`, forces *find* to measure times from the beginning of the current day instead of in 24-hour multiples. This frees you from dependence on the current time when you run *find*.

Directory Control

Early versions of *find* didn't give you much control over which directories *find* visited. However, once `-prune` was added, you could write an expression to keep *find* from descending into certain directories. For instance, to keep from descending into the *./src* subdirectory, you can do something like this:

```
find . -path ./src -prune -o …
```

And to skip *all* directories named *lib* (and all of their subdirectories):

```
find . -name lib -prune -o …
```

The `-prune` action is good for avoiding certain directories, but — without the regular expression tests added later, at least — it's not so good for limiting searches to a particular depth. In particular, it may not be obvious how to process only the entries in the current directory without any recursion. The answer with `-prune` is…

```
find . \( -type d ! -name . -prune \) \
   -o …
```

… which "prunes" all directories except the current directory `.` ("dot").

The new `-mindepth` and `-maxdepth` options make searching nested directories a lot easier. Use `-maxdepth n` to descend no more than *n* levels below the starting points given as command-line arguments. Hence, the option `-maxdepth 0` tells *find* to evaluate only the command-line arguments.

Conversely, `-mindepth n` tells *find* to ignore the first *n* levels of subdirectories. So, `-mindepth 1` processes all files *except* the command-line arguments. For instance, `find subdir -mindepth 1 -ls` descends into any directories found in `subdir` and lists those contents, but won't list `subdir` itself.

The `-depth` option has been in quite a few versions of *find;* it's not as "new" as some of the other features. It's not related to `-maxdepth` or `-mindepth`, though. The sidebar "Diving into the Depth Option" has more information about how this option is used.

The new `-fprintf` and `-fprint0` write results to a file

One "new" addition that's in a lot of versions of *find* is `-xdev` or `-mount`. (GNU *find* understands both options.) The options tell *find* to not descend into directories mounted from other filesystems. This is handy, for example, to avoid network-mounted filesystems.

A more specific test is `-fstype`, which tests true if a file is on a certain type of filesystem. For instance, `!-fstype nfs` is true for a file that's not on an *NFS*-type filesystem. Different systems have different filesystem names and types, though. To get a listing of what's on your system, use the new `-printf` action with the `%F` format directive to display the filesystems from the second field of each */etc/mtab* entry:

```
$ find `cut -d" " -f2 /etc/mtab` \
   -maxdepth 0 -printf "%-20p type %F\n"
/                    type ext3
/proc                type proc
/dev/pts             type devpts
/dev/shm             type tmpfs
...
```

(You'll probably find that same data in the second and third fields of each entry in */proc/mounts*.)

Text Output

Early versions of *find* effectively had one choice for outputting a pathname: print it to the standard output. Later, `-ls` was added to generate output similar to `ls -l`.

The new `-printf` action lets you use a C `printf()`-like format. The new format includes the usual format specifiers like the filename and the last-modification date, but it also has others specific to *find*. For instance, `%H` tells you which command-line argument *find* was processing when it found the current entry. One simple use for this is to make your own version of *ls* that gives just the information you want.

As an example, the following *bash* function, `findc()`, searches the command-line arguments (or, if there are no arguments, the current directory, `.`, instead) and prints information about all filenames ending with *.c*:

```
findc()
{
  find "${@-.}" -name '*.c' -printf \
    'DEPTH %2d  GROUP %-10g  NAME %f\n'
}
```

(*stat* might be simpler to use if you want a recursive listing and if *stat*' s format specifiers give the information you want.)

The new `-mindepth` and `-maxdepth` options make searching nested directories a lot easier

The longstanding `-print` action writes a pathname to the standard output, followed by a newline character. If that pathname happens to contain a newline, you get two newlines. (A newline is legal in a filename.) Most shells also break command-line arguments into words at whitespace (tabs, spaces and newlines), meaning that command substitution (the backquote operators) could fail if, say, a filename contained spaces. It wasn't too long before programmers fixed this problem by adding the `-print0` action. It outputs a pathname followed by NUL (a zero byte). Because NUL isn't legal in a filename, this pathname delimiter solved the problem — when *find* output was piped to the command `xargs -0`, which accepts NUL as an argument separator.

Because *find* can do many different tests as it traverses a filesystem, it's good to be able to choose what should be done in each individual case. For instance, if you run a nightly *cron* job to clean up various files and directories from all of your disks, it's nice to do all of the tests in a single pass through the filesystem instead of making a complete pass for each of your tests. But it's also good to avoid the overhead of running utilities like *rm* and *rmdir* over and over, once per file, in a *find* job like this one using `-exec`:

```
find /var/tmp -mtime +3 \( \
  \( -type f -exec rm -f {} \; \) -o \
  \( -type d -exec ..... {} \; \) \
\)
```

This inefficiency could be solved by replacing `-exec` with `-print` or `-print0`, then piping *find*'s output to *xargs*. *xargs* collects arguments and passes them to another program each time it has collected "enough." But all the text from `-print` or `-print0` goes to *find*'s standard output, so there's been no easy way to tell which pathnames were from which test (which are files, which are directories, etc...).

The new `-fprintf` and `-fprint0` actions can solve this problem. They write a formatted string to a file you specify. For instance, the following example writes a NUL-separated list of the files from */var/tmp/* into the file named by `$files` and a list of directories into the file named by `$dirs`:

```
dirs=`mktemp`
files=`mktemp`
find /var/tmp \( \
  \( -type f -fprint0 $files \) -o \
  \( -type d -fprint0 $dirs \) \
\)
```

Other New Tests

The `-empty` test is true for an empty file or directory. (An empty file has no bytes; an empty directory has no entries.) It's handy for removing empty directories while you're cleaning a filesystem. If you also use `-depth`, all of the files in a directory should be removed before *find* examines the directory itself. Then you can use an expression like the following:

```
find /tmp -depth \( \
  \( -mtime +3 -type f -exec rm -f {} \; \) \
  -o \( -type d -empty -exec rmdir {} \; \) \
\)
```

The `-false` "test" is always false, and `-true` is always true. These are a lot more efficient than the old methods (`-exec false` and `-exec true`) that execute the external Linux commands *false* and *true*.

The `-perm` test has long accepted arguments like `-perm 222` (which means "exactly mode 222" or write-only) and `-perm -222` (which means "all of the write mode bits are set"). Now, `-perm` also accepts arguments starting with a *plus sign*. It means "any of these bits are set." For instance, `-perm +222` is true when any write bit is set.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for 25 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.