Searching Recover files from obscure places by searching for their contents

Files: find what

Ever mislaid an item in Linux? That mayonnaise recipe? Emails to Auntie Gwen? That huge file you downloaded? Fear not – **Dr Chris Brown** talks you through it.

Our expert

Dr Chris Brown is a freelance Linux instructor with a PhD in particle physics as well as Novell CLP and Red Hat RHCE certification. He has also written a book on SUSE Linux for O'Reilly.

One way of estimating the relative importance of the tasks that folk use Linux for would be to count the number of different applications that have been written to perform each of those tasks. Given the rather large number of programs that exist for "finding stuff", we might conclude that the thing users do most often is to lose it in the first place! In this tutorial we'll survey a range of applications that allow us to search for files and other data by name, attributes, or content.

Searches based on file name

The simplest kinds of search are those based on file name, and the shell's filename wildcard matching provides a starting point for this. For example, the command

```
$ ls *invoice*
```

will list all file names in the current directory containing the string **invoice**. Not too impressive? Why not try something like:

```
$ ls */*invoice*
```

which will list files with **invoice** in the name in any subdirectories of your current directory? Then you can extend the idea to whatever level you want, maybe using something like this:

```
$ ls  *invoice*  */*invoice*  */*/*invoice*
```

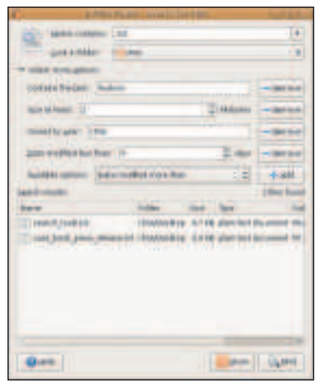
If you want to search the entire file system for a file based on a file name, the **slocate** command provides a solution. For example,

```
$ slocate invoice
```

will find all files with names that contain the string **invoice**. You'll find that **slocate** is lightning fast because it uses a pre-built index of filenames. This index is built using the program *updatedb* (the **slocate** command with the **-u** option does the same thing) which is usually run once a day via *cron* or *anacron*. On my Ubuntu 7.04 distribution, the *slocate* database is **/var/lib/slocate/slocate.db**. This is the only down-side of *slocate* – it won't find files that were created since *updatedb* was last run.

S for secure

In case you were wondering, the **s** in **slocate** stands for 'secure'. Here's the scoop on this: the *updatedb* program (the one that builds the index) runs with root privilege, so it can be sure of seeing all the files. This means that potentially there will be files listed in the **slocate.db** index that ordinary users should not be able to see. These might be system files, or they might be private files belonging to other users. The *slocate* index also keeps a record of the ownership and permissions on the files, and the *slocate* program is careful not to show you file names that you shouldn't be able to see. There was (I think) an older program called *locate* that wasn't this smart, but on a modern Linux distribution, **slocate** and **locate** are links to the same program.



› Some search criteria have been added in this example, but you can add lots more if you need to.

Don't give me bad news...

Among the output from find you'll often notice a bunch of error messages relating to directories we don't have permission to search. Sometimes, there can be so many of these messages that they entirely swamp the 'good' output. You can easily suppress the error messages by redirecting them to the 'black hole' device, **/dev/null**. To do this, simply append **2> /dev/null** to the command line.

Specialised search: which and whereis

There are a couple of more specialised search tools, *whereis* and *which*, that should be mentioned for the sake of completeness. The program whereis searches for the executable, source code and documentation (manual page) for a specified command. It looks in a pre-defined list of directories. For example:

```
$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

tells us the location the executable (binary) and the man page for the **ls** command. The **which** command is even more specialised. It simply looks up a specified command on our search path, reporting where it would first find it. For example:

```
$ which vi
/usr/bin/vi
```

tells us that the **vi** command is in **/usr/bin/vi**. Effectively, this command answers the question "If I entered the command **vi**, which program would actually get run?"

Searching on steroids: find

At the other end of the scale is the top-of-the-range search tool, find. In addition to filename-based searching, find is able to locate files based on ownership, access permissions, time of last access, size, and much else besides. Of course, the price you pay for all this flexibility is a rather perplexing command syntax. We'll dive into the details later, but here's an example to give you the idea:

```
$ find /etc -name '*.conf' -user cupsys -print
find: /etc/ssl/private: Permission denied
find: /etc/cups/ssl: Permission denied
/etc/cups/cupsd.conf
/etc/cups/printers.conf
```

In this example, find is searching in (and below) the directory **/etc** for files whose name ends in **.conf** and that are owned by the **cupsys** account.

you're looking for

Generally, the syntax of the **find** command is of the form:

```
$ find <where to look> <what to look for> <what to do with it>
```

The "where to look" part is simply a space-separated list of the directories we want find to search. For each one, find will recursively descend into every directory beneath those specified. Our table overpage, titled *Search criteria for find* lists the most useful search criteria (the "what to look for" part of the command), while the smaller table *Actions for find* on page 93 lists the most useful actions (the "what to do with it" part of the command). Neither of these is a complete list, so check the manual page for the full story. If no other action is specified, the **-print** action is assumed, with the result that the pathname of the selected file is printed (or to be more exact, written to standard output). This is a very common use of **find**. I should perhaps point out that many of the search criteria supported by **find** are really intended to help in rounding up files to perform some administrative operation on them (make a backup of them, perhaps) rather than helping you find odd files you happen to have mislaid.

Learning by Example

It takes a while to get your head around all this syntax, so maybe a few examples would help ...

Example 1 This is a simple name-based search, starting in my home directory and looking for all *PowerPoint* (**.ppt**) files. Notice we've put the filename wildcard expression in quotes to stop the shell trying to expand it. We want to pass the argument **'*.ppt'** directly and let find worry about the wildcard matching.

```
$ find ~ -name '*.ppt'
```

Example 2 You can supply multiple "what to look for" tests to find and by default they will be logically AND-ed, that is, they must all be true in order for the file to match. Here, we look for directories under **/var** that are owned by daemon:

```
$ find /var -type d -user daemon
```

Example 3 This shows how you can OR tests together rather than AND-ing them. Here, we're looking in **/etc** for files that are either owned by the account **cupsys** or are completely empty:

```
$ find /etc -user cupsys -or -size 0
```

Example 4 This uses the '**!**' operator to reverse the sense of a test. Here, we're searching **/bin** for files that aren't owned by **root**:

```
$ find /usr/bin ! -user root
```

Example 5 The tests that make numeric comparisons are especially confusing. Just remember that '**+**' in front of a number means 'more than', '**-**' means 'less than', and if there is no '**+**' or '**-**', **find** looks for an exact match. These three example search for files that have been modified less than 10 minutes ago, more than 1 year ago, and exactly 4 days ago. (This third example is probably not very useful.)

```
$ find ~ -mmin -10
$ find ~ -mtime +365
$ find ~ -mtime 4
```

Why is this not a command?

The **which** command can – occasionally – give a misleading answer, if the command in question also happens to be a built-in command of the *bash* shell. For example:

```
$ which kill
/bin/kill
```

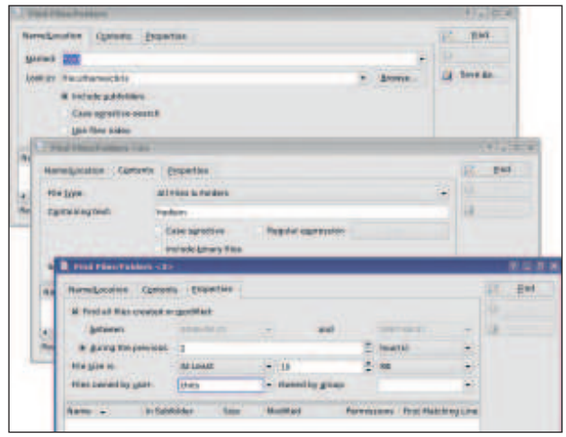
tells us that the **kill** command lives in **/bin**. However, **kill** is also a built-in *bash* command, so if I enter a command like

```
$ kill -HUP 1246
```

it will actually run the shell's built-in **kill** and not the external command.

To find out whether a command is recognised as a shell built-in, an alias, or an external command, you can use the **type** command, like this:

```
$ type kill
kill is a shell builtin
```



› **According to my experiments, *kfind* does not rely on running *find* behind the scenes.**

Example 6 Perhaps the most confusing tests of all are those made on a file's access permissions. This example isn't too bad, it looks for an exact match on the permissions **644** (which would be represented symbolically by **ls -l** as **rw-r—r--**:

```
$ find ~ -perm 644
```

Example 7 Here, we look for files that are writeable by anybody (that is, either the owner, the group, or rest-of-world). The two examples are equivalent; the first uses the traditional octal notation, the second uses the same symbolic notation for representing permissions that *chmod* uses:

```
$ find ~ -perm -222
$ find ~ -perm -ugo=w
```

Example 8 Here, we look for files that are writeable by everybody (that is, by the owner and the group and the rest-of-world):

```
$ find ~ -perm /222
$ find ~ -perm /ugo=w
```

Example 9 So far we've just used the default **-print** action of **find** to display the names of the matching files. Here's an example that »

Quick tip

The *Google Desktop 1.0* search tool is available for Linux, and needs requires *glibc 2.3.2+* and *gtk+ 2.2.0+* to run. We gave it 6/10 in **LXF97**'s review, as we thought it still has some way to go. Download it and judge for yourself!

Search criteria for find

Syntax	Description	Example
-name string	File name matches string (wildcards are allowed)	**-name '*.jpg'**
-iname string	Same as **-name** but not case sensitive	**-iname '*tax*'**
-user username	File is owned by username	**-user chris**
-group groupname	File has group groupname	**-group admin**
-type x	File is of type '**x**', one of: **f** – regular file **d** – directory **l** – symbolic link **c** – character device **b** – block device **p** – named pipe (FIFO)	**-type d**
-size +N	File is bigger than **N** 512-byte blocks (use suffix **c** for bytes, **k** for kilobytes, **M** for megabytes)	**-size +100M**
-size -N	File is smaller than **N** blocks (use suffix **c** for bytes, **k** for kilobytes, **M** for megabytes)	**-size -50c**
-mtime -N	File was last modified less than **N** days ago	**-mtime -1**
-mtime +N	File was last modified more than **N** days ago	**-mtime +14**
-mmin -N	File was last modified less than **N** minutes ago	**-mmin -10**
-perm mode	The files permissions exactly match mode. The mode can be specified in octal, or using the same symbolic notation that *chmod* supports	**-perm 644**
-perm -mode	All of the permission bits specified by mode are set.	**-perm -ugo=x**
-perm /mode	Any of the permission bits specified by mode is set	**-perm /011**

» uses the **-exec** option to move all matching files into a backup directory. There are a couple of points to note here. First, the notation **{}** gets replaced by the full pathname of the matching file, and the '**;**' is used to mark the end of the command that follows **-exec**. Remember: '**;**' is also a shell metacharacter, so we need to put the backslash in front to prevent the shell interpreting it.

```
$ find ~ -mtime +365 -exec mv {} /tmp/mybackup \;
```

Never mind the file name, what's in the file?

As we've seen, tools such as find can track down files based on file name, size, ownership, timestamps, and much else, but find cannot select files based on their content. It turns out that we can do some quite nifty content-based searching using **grep** in conjunction with the shell's wildcards. This example is taken from my personal file system:

```
$ grep -l Hudson */*
Desktop/suse_book_press_release.txt
google-earth/README.linux
Mail/inbox.ev-summary
Mail/sent-mail.ev-summary
snmp_training/enterprise_mib_list
```

Here, we're asking grep to report the names of the files containing a match for the string **Hudson**. The wildcard notation ***/*** is expanded by the shell to a list of all files that are one level below the current directory. If we wanted to be a bit more selective on the file name, we could do something like:

```
$ grep -l Hudson */*.txt
Desktop/search_tools.txt
Desktop/suse_book_press_release.txt
```

which would only search in files with names ending in **.txt**. In principal you could extend the search to more directory levels, but in practice you may find that the number of file names matched by the shell exceeds the number of arguments that can appear in the argument list, as happened when I tried it on my system:

```
$ grep -l Hudson  */*  */*/*
bash: /bin/grep: Argument list too long
```

A more powerful approach to content-based searching is to use

The truth about find

The individual components of a **find** command are known as expressions, (or more technically, as predicates). For example, **-uname cupsys** is a predicate. The find command operates by examining each and every file under the directory you ask it to search and evaluating each of the predicates in turn against that file. Each predicate returns either true or false, and the results of the predicates are logically AND-ed together. If one of the predicates returns a false result, find does not evaluate the remaining predicates. So for example in a command such as:

```
$ find . -user chris -name '*.txt' -print
```

if the predicate **-user chris** is false (that is, if the file is not owned by **chris**) find will not evaluate the remaining predicates. Only if **-user chris** and **-name '*.txt'** both return true will find evaluate the **-print** predicate (which writes the file name to standard output and also returns the result 'true').

› **Searching for 'Linux Format' on the Gnome desktop.**

In Ireland, the inhabitants of Kerry are pleased to be associated with something this smart, for a change...

grep in conjunction with **find**. This example shows a search for files under my home directory ('**~**') whose names end in **.txt**, that contain the string **Hudson**.

```
$ find ~ -name '*.txt' -exec grep -q Hudson {} \; -print
/home/chris/Desktop/search_tools.txt
/home/chris/Desktop/suse_book_press_release.txt
```

This approach does not suffer from the argument list overflow problem that our previous example suffered from. Remember, too, that find is capable of searching on many more criteria that just file name, and **grep** is capable of searching for regular expressions not just fixed text, so there is a lot more power here than this simple example suggests. If you're unclear about the syntax of this example, read *The truth about find*, below left. In this example, the predicate **-exec grep -q Hudson {} \;** returns true if **grep** finds a match for the string **Hudson** in the specified file, and false if not. If the predicate is false, **find** does not continue to evaluate any following expressions, that is, it does not execute the **-print** action.

Graphical tools

So far we've focused on command line search tools. Of course there are graphical tools too. Gnome includes a graphical tool called *gnome-search-tool*, that's shown on page 90. When it starts up, this tool presents a minimal interface that simply allows you to specify a partial match on the file name, and a directory to search, but you can progressively add more search criteria, some of which are shown in the figure. These criteria will look familiar from our previous discussion of **find**, and in fact *gnome-search-tool* runs find in the background to do the actual search. How do I know? Well, because we tried renaming the *find* executable, and discovered that *gnome-search-tool* subsequently fails with the error "Failed to execute child process 'find'."

The KDE desktop has a similar tool called *kfind*, but it takes a slightly different approach to its user interface, splitting the search criteria across three tabs which are shown in page 91.

Computers vs Humans

Though programs like *find* and *Beagle* seem impressive, we are a long way from having computer-based search tools that mimic the capabilities of humans. We cannot ask the computer, for example, "Where is that picture I took of the cows on the beach?", unless, of course, I had carefully named the file **cows_on_beach.jpg**. Nor can we ask, say, for all the mp3 files that feature solo performances on the cello. So to all you bright young programmers out there – c'mon, what are you messing about at? Get coding!

Actions for find

Action	Description
-print	Print the full pathname of the file to standard output
-ls	Give a full listing of the file, equivalent to running **ls -dils**
-delete	Delete the file
-exec command	Execute the specified command. All following arguments to find are taken to be arguments to the command until a '**;**' is encountered. The string **{}** is replaced by the current file name.

Let beagle sniff out your data.

Named after the hound famous for its keen sense of smell and tracking instinct, *Beagle* is in a different league from the other search tools. To quote *Beagle*'s home page (**http://beagle-project.org**) "*Beagle* is a search tool that ransacks your personal information space to find whatever you're looking for". It's capable of finding text in a wide variety of document types including plain text, *OpenOffice.org* and *Microsoft Office* documents, PDF files, HTML, manual pages, and in various other information sources such as *Evolution* and *KMail* mail folders, address books, notes made in *Tomboy* and *knotes*, and RSS feeds. (For a complete list, see **http://beagle-project.org/Supported_Filetypes**)

Beagle is a .NET app that needs the Mono runtime plus quite a few other libraries. Eighteen months ago, I struggled with a myriad of dependency and versioning problems to get a working version of *Beagle* for a book I was writing. Things seem to have matured, with most modern Linux distros offering a version of *Beagle* that works "out of the box". On some distros, *Beagle* is integrated with Gnome. The screenshot below left shows it running on an Ubuntu 7.04 desktop. Here, I have searched for the phrase "Linux Format" and turned up quite a number of hits within the file system and within my mail archive. For KDE desktops, there is a graphical front-end called *Kerry Beagle*; the screenshot at the top of this page shows it running on a SUSE Linux system. In case you were wondering, the *Kerry Beagle* is also a type of hunting hound, with a name that (conveniently for the KDE guys) begins with 'K'.

Beagle uses pre-built indexes to offer fast searching, but the experience is far more dynamic than the once-per-day indexing using by the *slocate* program we saw earlier. The first time you run *Beagle*, it has to crawl your home directory and index all of your data. If you have a lot of files, emails, or other documents, or your system is under heavy load, this can take up to several hours, and you may have to wait for all of your data to be indexed. *Beagle* also uses the *inotify* feature built into modern Linux kernels to update its index automatically when things change in the file system. The *beagled* daemon that carries out the indexing process runs under your normal user account (not as root) and is restricted to indexing material in your home directory – *Beagle* is very much a tool for searching your private space, not the system files. The indexing process deliberately throttles back on its use of CPU resources to avoid loading the machine too heavily. Do be aware, though, that the *Beagle* indexes can take up a significant amount of space. The *Beagle* FAQ suggests that the index uses 5-10 per cent of the size of the data being indexed, but my own system the figure is about 2 per cent (a 71 MB index for a 3.6 GB file system). The indexes are stored in an extensive hierarchy of files under the directory **~/.beagle**.

Though *Beagle* is most likely to be accessed through a GUI, there are command line tools available, in particular *beagle-query*, which performs command line searches. There are also some handy administrative tools, including *beagle-config*, which configures the beagle indexing process; and *beagle-status*, that gives a regularly updated display of the progress of the indexing daemon *beagled*.

Other apps

Searchmonkey is based on *Gtk+*,used for matching files and contents using regular expressions. **http://searchmonkey.sourceforge.net**.

Strigi is a light-weight, desktop-independent search daemon to extract data from within files *eg* the length of audio clips, contents of a document, or resolution of image files. **http://strigi.sourceforge.net**.

Tracker lets users search documents in a similar way to *Spotlight* in OS X. **www.gnome.org/projects/tracker**.