Find

NetBrain
LEARN MORE
Introducing Dynamic Mapping
Say goodbye to static network diagrams

Check my Unix Page and My blog

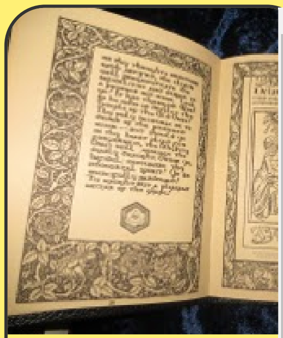

Table of Contents

Original version written in 1995 and published in the Sun Observer

A complete tutorial on find. How to search for a file and do something with it. How to prevent find from

awk
Reference
HTML
Magic
Search
About
Donate NEW

searching certain directories.

Introduction to find

One of the commands that everyone should master is the **find** command. The first, and most obvious, use is **find**'s ability to locate old, big, or unused files, or files that you forgot where they are.

The other important characteristic is **find**'s ability to travel down subdirectories. If you wanted a recursive directory list, and **ls** doesn't have this option, use **find**.

Problems with other methods

Normally the shell provides the argument list to a command. That is, Unix programs are frequently given file names and not directory names. Only a few programs can be given a directory name and march down the directory searching for subdirectories. The programs **find**, **tar**, **du**, and **diff** do this. The commands **chmod**, **chgrp**, **rm**, and **cp** will, but only if a **-r** or **-R** option is specified. In SunOS 4.1, **ls** supports the **-R** recursive option.

In general, most commands do not understand directory structures, and rely on the shell to expand meta-characters to directory names. That is, if you wanted to delete all object files ending with a ".o" in a group of directories, you could type

```
rm *.o */*.o */*/*.o
```

Not only is this tedious to type, it may not find all of the files you are searching for. The shell has certain blind spots. It will not match files in directories starting with a period. And, if any files match ***/*/*/*.o**, they would not be deleted.

Another problem, although one that occurs less frequently in SunOS 4.0 and above, is typing the above, and getting the error "Arguments too long." This means the number of arguments on the list was too large for the shell.

The answer to these problems is using **find**.

The Simplest Example

A simple example of find is using it to print the names of all of the files in the directory and all subdirectories. This is done with the simple command

```
find . -print
```

The first argument to **find** is a directory or file name. The arguments after the path names always start with a minus sign, and tells find what to do once it finds a file. These are the search options. In this case, the file name is printed. Any number of directories can be specified. You can use the tilde character supported by the C shell, as well as particular paths.

```
find ~ ~barnett /usr/local -print
```

And if you have a very slow day, you can type

```
find / -print
```

which will list every file on the system.

Excuse me. I just spoke an untruth. Every directory always contains the files **.** and **..**. **Find** could report these obvious files, but thankfully doesn't.

Using find with other commands

Find sends its output to standard output. One way to use this is with the back quote command substitution:

```
ls -ld `find . -print`
```

The **find** command is executed, and its output replaces the back quoted string. **Ls** sees the output of **find**, and doesn't even know **find** was used.

Using xargs with find

An alternate method uses the **xargs** command. **Xargs** and **find** work together beautifully. **Xargs** executes its arguments as commands, and reads standard input to specify arguments to that command. **Xargs** knows the maximum number or arguments each command line can handle, and does not exceed that limit. While the command

```
ls -ld `find / -print`
```

might generate an error when the command line is too large, the equivalent command using **xargs** will never generate that error:

```
find / -print | xargs ls -ld
```

Looking for files with particular names

You can look for particular files by using a regular expression with meta-characters as an argument to the **-name** option. The shell also understands these meta-characters, as it also understands regular expressions. It is necessary to quote these special characters, so they are passed to **find** unchanged. Either the back quote or single quotes can be used:

```
find . -name *.o -print
find . -name '*.o' -print
find . -name '[a-zA-Z]*.o' -print
```

The path of the file is not matched with the **-name** option, merely the name in the directory without the path leading to the file.

Looking for files by type

If you are only interested in files of a certain type, use the **-type** argument, followed by one of the following characters:

Character	Meaning
b	Block special file (see mknode(8))
c	Character special file (see mknode(8))
d	Directory
f	Plain file
p	Named Pipe File
l	Symbolic link
s	Socket

Unless you are a system administrator, the important types are directories, plain files, or symbolic links (i.e. types **d**, **f**, or **l**).

Using the **-type** option, another way to recursively list files is:

```
find . -type f -print | xargs ls -l
```

It can be difficult to keep track of all of the symbolic links in a directory. The next command will find all of the symbolic links in your home directory, and print the files your symbolic links point to.

```
find . -type l -print | xargs ls -ld | awk '{print $10}'
```

Looking for files by sizes

Find has several options that take a decimal integer. One such argument is **-size**. The number after this argument is the size of the files in disk blocks. Unfortunately, this is a very vague number. Earlier versions of Unix used disk blocks of 512 bytes. Newer versions allow larger block sizes, so a "block" of 512 bytes is misleading.

This confusion is aggravated when the command **ls -s** is used. The **-s** option lists the size of the file in blocks. If the command is "/usr/bin/ls," the block size is 1024 bytes. If the command is "/usr/5bin/ls," the block size is 512 bytes.

Let me confuse you some more. The following shows the two versions of ls:

```
% /usr/bin/ls -sl file
14 -rwxr-xr-x 1 barnett 13443 Jul 25 23:27 file
% /usr/5bin/ls -sl file
28 -rwxr-xr-x 1 barnett staff 13443 Jul 25 23:27 file
```

Can you guess what block size should be specified so that **find** prints this file? The correct command is:

```
find . -size 27 -print
```

because the actual size is between 26 and 16 blocks of 512 bytes each. As you can see, "ls -s" is not an accurate number for **find**. You can put a **c** after the number, and specify the size in bytes,

To search for files using a range of file sizes, a minus or plus sign can be specified before the number. The minus sign means "less than," and the plus sign means "greater than." This next example lists all files that are greater than 10,000 bytes, but less than 32,000 bytes:

```
find . -size +10000c -size -32000c -print
```

When more than one qualifier is given, both must be true.

Searching for old files

If you want to find a file that is 7 days old, use the **-mtime** option:

```
find . -mtime 7 -print
```

An alternate way is to specify a range of times:

```
find . -mtime +6 -mtime -8 -print
```

Mtime is the last modified time of a file. You can also think of this as the creation time of the file, as Unix does not distinguish between creation and modification. If you want to look for files that have not been used, check the access time with the **-atime** argument. A command to list all files that have not be read in thirty days or more is

```
find . -type f -atime +30 -print
```

It is difficult to find directories that have not been accessed because the **find** command modifies the directory's access time.

There is another time associated with each file, called the **ctime**, accessed with the **-ctime** option. This will have a more recent value if the owner, group, permission or number of links is changed, while the file itself does not. If you want to search for files with a specific number of links, use the **-links**option.

Searching for files by permission

Find can look for files with a specific permission. It uses an octal number for these permissions. The string **rw-rw-r--**, indicates you and members of your group have read and write permission, while the world has read only priviledge. The same permissions, when expressed as an octal number, is **664**. To find all "*.o" files with the above permission, use:

```
find . -name *.o -perm 664 -print
```

If you want to see if you have any directories with world write permission, use:

```
find . -type d -perm 777 -print
```

This only matches the exact combination of permissions. If you wanted to find all directories with group write permission, there are several combinations that can match. You could list each combination, but **find** allows you to specify a pattern that can be bit-wise **AND**ed with the permissions of the file. Simply put a minus sign before the octal value. The group write permission bit is octal 20, so the following negative value:

```
find . -perm -20 -print
```

will match the following common permissions:

```
+------------------------+
|Permission   Octal value |
+------------------------+
|rwxrwxrwx    777         |
|rwxrwxr-x    775          |
|rw-rw-rw-    666          |
|rw-rw-r--    664          |
|rw-rw----    660          |
+-------------------------+
```

If you wanted to look for files that you can execute, (i.e. shell scripts or programs), you want to match the pattern "--x------," by typing:

```
find . -perm -100 -print
```

When the **-perm** argument has a minus sign, all of the permission bits are examined, including the set user ID bits.

Owners and groups

Often you need to look for a file that has certain permissions and belonging to a certain user or group. This is done with the **-user** and **-group** search options. To find all files that are set user ID to root, use:

```
find . -user root -perm -4000 -print
```

To find all files that are set group ID to staff, use:

```
find . -group staff -perm -2000 -print
```

Instead of using a name or group in **/etc/passwd** or **/etc/group**, you can use a number:

```
find . -user 0 -perm -4000 -print
find . -group 10 -perm -2000 -print
```

Often, when a user leaves a site, their account is deleted, but their files are still on the computer. A system manager can use the **-nouser** or **-nogroup** to find files with an unknown user or group ID.

Find and commands

So far, after **find** has found a file, all it has done is printed the filename. It can do much more than that, but the syntax can get hairy. Using **xargs** saves you this mental effort, but it isn't always the best solution.

If you want a recursive listing, **find**'s output can be piped into **| xargs ls -l** but it is more efficient to use the built in **-ls** option:

```
find . -ls
```

This is similar to the command:

```
find . -print | xargs ls -gilds
```

You could also use **ls -R** command, but that would be too easy.

Find and Cpio

Find also understands **cpio**, and supports the **-cpio** and **-ncpio** commands:

```
find . -depth -cpio >/dev/rmt0
find . -depth -ncpio >/dev/rmt0
```

which do the same as

```
find . -depth -print | cpio -oB >/dev/rmt0
find . -depth -print | cpio -ocB >/dev/rmt0
```

Using Find to Execute Commands

I have already discussed how to use **xargs** to execute commands. **Find** can execute commands directly. The syntax is peculiar, which is one reasons I recommend **xargs**. The syntax of the **-exec** option allows you to execute any command, including another find command. If you consider that for a moment, you realize that **find** needs some way to distinguish the command it's executing from its own arguments. The obvious choice is to use the same end of command character as the shell (i.e. the semicolon). Since the shell normally uses the semicolon itself, it is necessary to "escape" the character with a backslash or quotes. There is one more special argument that **find** treats differently: **{}**. These two characters are used as the variable whose name is the file **find** found. Don't bother re-reading that last line. An example will clarify the usage. The following is a trivial case, and uses the **-exec** option to mimic the **"-print'** option.

```
find . -exec echo {} ;
```

The C shell uses the characters **{** and **}**, but doesn't change **{}**, which is why it is not necessary to quote these characters. The semicolon must be quoted, however. Quotes can be used instead of a backslash:

```
find . -exec echo {} ';'
```

as both will pass the semicolon past the shell to the **find** command. As I said before, **find** can even call **find**. If you wanted to list every symbolic link in every directory owned by group "staff" you could execute:

```
find `pwd` -group staff -exec find {} -type l -print ;
```

To search for all files with group write permission, and remove the permission, you can use

```
find . -perm -20 -exec chmod g-w {} ;
```

or

```
find . -perm -20 -print | xargs chmod g-w
```

The difference between **-exec** and **xargs** are subtle. The first one will execute the program once per file, while **xargs** can handle several files with each process. However, **xargs** may have problems with files that contain embedded spaces.

Occasionally people create a strange file that they can't delete. This could be caused by accidentally creating a file with a space or some control character in the name. **Find** and **-exec** can delete this file, while **xargs** could not. In this case, use **ls -il** to list the files and i-nodes, and use the **-inum** option with **-exec** to delete the file:

```
find . -inum 31246 -exec rm [] ';'
```

If you wish, you can use **-ok** which does the same as **-exec**, except the program asks you first to confirm the action before executing the command. It is a good idea to be cautious when using **find**, because the program can make a mistake into a disaster. When in doubt, use **echo** as the command. Or send the output to a file and examine the file before using the file as input to **xargs**. This is how I discovered that **find** can only use one **{}** in the arguments to **-exec**. I wanted to rename some files using "-exec mv {} {}.orig" but I learned that I have to write a shell script that I told **find** to execute.

File comparisons

Whenever I upgraded to a new version of Unix, one common problem was making sure I maintained all of the changes made to the standard release of Unix. Previously, I did a **ls -lt** in each directory, and then I examined the modification date. The files that were changed has an obviously newer date that the original programs. Even so, finding every change was tedious, as there were dozens of directories to be searched.

A better solution is to create a file as the first step in upgrading. I usually call this **FirstFile**. **Find** has a **-newer** option that tests each file and compares the modification date to the newer file. If you then wanted to list all files in **/usr** that need to be saved when the operating system is upgraded, use:

```
find /usr -newer /usr/FirstFile -print
```

This could then be used to create a **tar** or **cpio** file that would be restored after the upgrade.

Expressions

Find allows complex expressions. To negate a test, put a **!** before the option. Since the C shell interprets this command, it must be escaped. To find all files the same age or older than "FirstFile," use

```
find /usr ! -newer /FirstFile -print
```

The "and" function is performed by the **-a** option. This is not needed, as two or more options are **AND**ed automatically. The "or" function is done with the **-o** option. Parenthesis can be used to add precedence. These must also be escaped. If you wanted to print object and "a.out" files that are older than 7 days, use:

```
find . ( -name a.out -o -name *.o ) -print
```

Keeping find from going too far

The most painful aspect of a large NFS environment is avoiding the access of files on NFS servers that are down. **Find** is particularly sensitive to this, because it is very easy to access dozens of machines with a single command. If **find** tries to explore a file server that happens to be down, it will time out. It is important to understand how to prevent **find** from going too far.

The important option in this case is **-prune**. This option confuses people because it is always true. It has a side-effect that is important. If the file being looked at is a directory, it will not travel down the directory. Here is an example that lists all files in a directory but does not look at any files in subdirectories under the top level:

```
find * -type f -print -o -type d -prune
```

This will print all plain files and prune the search at all directories. To print files except for those in a Source Code Control Directories, use:

```
find . -print -o -name SCCS -prune
```

If the **-o** option is excluded, the SCCS directory will be printed along with the other files.

Another useful combination is using **-prune** with **-fstype** or **-xdev**. **Fstype** tests for file system types, and expects an argument like **nfs** or **4.2**. The later refers to the file system introduced in the 4.2 release of the Berkeley Software Distribution. To limit **find** to files only on a local disk or disks, use the clause **-fstype 4.2 -prune** or **-o -fstype nfs -prune**. If you needed to limit the search to one particular disk partition, use **-xdev**, The later is very useful if you want to help a congested disk partition, and wanted to look for all files greater than 40 blocks on the current disk partition;

```
find . -size -40 -xdev -print
```

Fast Find

There is a new feature in **find** that makes a dramatic improvement is searching for files. This typically only works if you are searching for a file on a local disk. The system manager must run a script called **updatedb** once a night using **cron**. This creates a database called **/usr/lib/find/find.codes**.

When **find** is executed with a single argument, it looks for the file using the database, and reports where the file is. You can create this database yourself, if you wish, as **find** will look for the database at any location the environment variable **FCODES** defines.

This document was translated by troff2html v0.21 on September 22, 2001 and then manually edited to make it compliant with: