Finding Files

Edition 4.2.33, for GNU `find` version 4.2.33
20 June 2015

by David MacKenzie and James Youngman

This file documents the GNU utilities for finding files that match certain criteria and performing various operations on them.

Table of Contents

1 Introduction

This manual shows how to find files that meet criteria you specify, and how to perform various actions on the files that you find. The principal programs that you use to perform these tasks are `find`, `locate`, and `xargs`. Some of the examples in this manual use capabilities specific to the GNU versions of those programs.

GNU `find` was originally written by Eric Decker, with enhancements by David MacKenzie, Jay Plett, and Tim Wood. GNU `xargs` was originally written by Mike Rendell, with enhancements by David MacKenzie. GNU `locate` and its associated utilities were originally written by James Woods, with enhancements by David MacKenzie. The idea for '`find -print0`' and '`xargs -0`' came from Dan Bernstein. The current maintainer of GNU findutils (and this manual) is James Youngman. Many other people have contributed bug fixes, small improvements, and helpful suggestions. Thanks!

To report a bug in GNU findutils, please use the form on the Savannah web site at `http://savannah.gnu.org/bugs/?group=findutils`. Reporting bugs this way means that you will then be able to track progress in fixing the problem.

If you don't have web access, you can also just send mail to the mailing list. The mailing list `bug-findutils@gnu.org` carries discussion of bugs in findutils, questions and answers about the software and discussion of the development of the programs. To join the list, send email to `bug-findutils-request@gnu.org`.

Please read any relevant sections of this manual before asking for help on the mailing list. You may also find it helpful to read the NON-BUGS section of the `find` manual page.

If you ask for help on the mailing list, people will be able to help you much more effectively if you include the following things:

- The version of the software you are running. You can find this out by running '`locate --version`'.
- What you were trying to do
- The *exact* command line you used
- The *exact* output you got (if this is very long, try to find a smaller example which exhibits the same problem)
- The output you expected to get

1.1 Scope

For brevity, the word *file* in this manual means a regular file, a directory, a symbolic link, or any other kind of node that has a directory entry. A directory entry is also called a *file name*. A file name may contain some, all, or none of the directories in a path that leads to the file. These are all examples of what this manual calls "file names":

```
parser.c
README
./budget/may-94.sc
fred/.cshrc
/usr/local/include/termcap.h
```

A *directory tree* is a directory and the files it contains, all of its subdirectories and the files they contain, etc. It can also be a single non-directory file.

These programs enable you to find the files in one or more directory trees that:

- have names that contain certain text or match a certain pattern;
- are links to certain files;
- were last used during a certain period of time;
- are within a certain size range;
- are of a certain type (regular file, directory, symbolic link, etc.);
- are owned by a certain user or group;
- have certain access permissions;
- contain text that matches a certain pattern;
- are within a certain depth in the directory tree;
- or some combination of the above.

Once you have found the files you're looking for (or files that are potentially the ones you're looking for), you can do more to them than simply list their names. You can get any combination of the files' attributes, or process the files in many ways, either individually or in groups of various sizes. Actions that you might want to perform on the files you have found include, but are not limited to:

- view or edit
- store in an archive
- remove or rename
- change access permissions
- classify into groups

This manual describes how to perform each of those tasks, and more.

1.2 Overview

The principal programs used for making lists of files that match given criteria and running commands on them are `find`, `locate`, and `xargs`. An additional command, `updatedb`, is used by system administrators to create databases for `locate` to use.

`find` searches for files in a directory hierarchy and prints information about the files it found. It is run like this:

```
find [file...] [expression]
```

Here is a typical use of `find`. This example prints the names of all files in the directory tree rooted in `/usr/src` whose name ends with '`.c`' and that are larger than 100 Kilobytes.

```
find /usr/src -name '*.c' -size +100k -print
```

Notice that the wildcard must be enclosed in quotes in order to protect it from expansion by the shell.

`locate` searches special file name databases for file names that match patterns. The system administrator runs the `updatedb` program to create the databases. `locate` is run like this:

```
locate [option...] pattern...
```

This example prints the names of all files in the default file name database whose name ends with '`Makefile`' or '`makefile`'. Which file names are stored in the database depends on how the system administrator ran `updatedb`.

```
locate '*[Mm]akefile'
```

The name `xargs`, pronounced EX-args, means "combine arguments." `xargs` builds and executes command lines by gathering together arguments it reads on the standard input. Most often, these arguments are lists of file names generated by `find`. `xargs` is run like this:

```
xargs [option...] [command [initial-arguments]]
```

The following command searches the files listed in the file `file-list` and prints all of the lines in them that contain the word '`typedef`'.

```
xargs grep typedef < file-list
```

1.3 `find` Expressions

The expression that `find` uses to select files consists of one or more *primaries*, each of which is a separate command line argument to `find`. `find` evaluates the expression each time it processes a file. An expression can contain any of the following types of primaries:

options affect overall operation rather than the processing of a specific file;

tests return a true or false value, depending on the file's attributes;

actions have side effects and return a true or false value; and

operators connect the other arguments and affect when and whether they are evaluated.

You can omit the operator between two primaries; it defaults to '`-and`'. See Section 2.11 [Combining Primaries With Operators], page 17, for ways to connect primaries into more complex expressions. If the expression contains no actions other than '`-prune`', '`-print`' is performed on all files for which the entire expression is true (see Section 3.1 [Print File Name], page 18).

Options take effect immediately, rather than being evaluated for each file when their place in the expression is reached. Therefore, for clarity, it is best to place them at the beginning of the expression.

Many of the primaries take arguments, which immediately follow them in the next command line argument to `find`. Some arguments are file names, patterns, or other strings; others are numbers. Numeric arguments can be specified as

`+n` for greater than *n*,

`-n` for less than *n*,

`n` for exactly *n*.

2 Finding Files

By default, `find` prints to the standard output the names of the files that match the given criteria. See Chapter 3 [Actions], page 18, for how to get more information about the matching files.

2.1 Name

Here are ways to search for files whose name matches a certain pattern. See Section 2.1.4 [Shell Pattern Matching], page 6, for a description of the *pattern* arguments to these tests.

Each of these tests has a case-sensitive version and a case-insensitive version, whose name begins with '`i`'. In a case-insensitive comparison, the patterns '`fo*`' and '`F??`' match the file names `Foo`, '`FOO`', '`foo`', '`fOo`', etc.

2.1.1 Base Name Patterns

-name *pattern* [Test]
-iname *pattern* [Test]

True if the base of the file name (the path with the leading directories removed) matches shell pattern *pattern*. For '`-iname`', the match is case-insensitive.[1] To ignore a whole directory tree, use '`-prune`' (see Section 2.9 [Directories], page 14). As an example, to find Texinfo source files in `/usr/local/doc`:

```
find /usr/local/doc -name '*.texi'
```

Notice that the wildcard must be enclosed in quotes in order to protect it from expansion by the shell.

As of findutils version 4.2.2, patterns for '`-name`' and '`-iname`' will match a file name with a leading '`.`'. For example the command '`find /tmp -name \*bar`' will match the file `/tmp/.foobar`. Braces within the pattern ('`{}`') are not considered to be special (that is, `find . -name 'foo{1,2}'` matches a file named `foo{1,2}`, not the files `foo1` and `foo2`.

2.1.2 Full Name Patterns

-wholename *pattern* [Test]
-iwholename *pattern* [Test]

True if the entire file name, starting with the command line argument under which the file was found, matches shell pattern *pattern*. For '`-iwholename`', the match is case-insensitive. To ignore a whole directory tree, use '`-prune`' rather than checking every file in the tree (see Section 2.9 [Directories], page 14). The "entire file name" as used by `find` starts with the starting-point specified on the command line, and is not converted to an absolute pathname, so for example `cd /; find tmp -wholename /tmp` will never match anything.

---

[1] Because we need to perform case-insensitive matching, the GNU fnmatch implementation is always used; if the C library includes the GNU implementation, we use that and otherwise we use the one from gnulib

-path *pattern* [Test]

-ipath *pattern* [Test]

These tests are deprecated, but work as for '-wholename' and '-iwholename', respectively. The '-ipath' test is a GNU extension, but '-path' is also provided by HP-UX find.

-regex *expr* [Test]

-iregex *expr* [Test]

True if the entire file name matches regular expression *expr*. This is a match on the whole path, not a search. For example, to match a file named ./fubar3, you can use the regular expression '.*bar.' or '.*b.*3', but not 'f.*r3'. See Section "Syntax of Regular Expressions" in *The GNU Emacs Manual*, for a description of the syntax of regular expressions. For '-iregex', the match is case-insensitive. There are several varieties of regular expressions; by default this test uses POSIX basic regular expressions, but this can be changed with the option '-regextype'.

-regextype *name* [Option]

This option controls the variety of regular expression syntax understood by the '-regex' and '-iregex' tests. This option is positional; that is, it only affects regular expressions which occur later in the command line. If this option is not given, GNU Emacs regular expressions are assumed. Currently-implemented types are

'emacs' Regular expressions compatible with GNU Emacs; this is also the default behaviour if this option is not used.

'posix-awk'
Regular expressions compatible with the POSIX awk command (not GNU awk)

'posix-basic'
POSIX Basic Regular Expressions.

'posix-egrep'
Regular expressions compatible with the POSIX egrep command

'posix-extended'
POSIX Extended Regular Expressions

Section 6.5 [Regular Expressions], page 49 for more information on the regular expression dialects understood by GNU findutils.

2.1.3 Fast Full Name Search

To search for files by name without having to actually scan the directories on the disk (which can be slow), you can use the locate program. For each shell pattern you give it, locate searches one or more databases of file names and displays the file names that contain the pattern. See Section 2.1.4 [Shell Pattern Matching], page 6, for details about shell patterns.

If a pattern is a plain string—it contains no metacharacters—locate displays all file names in the database that contain that string. If a pattern contains metacharacters, locate only displays file names that match the pattern exactly. As a result, patterns that contain metacharacters should usually begin with a '*', and will most often end with one as well. The exceptions are patterns that are intended to explicitly match the beginning or end of a file name.

If you only want `locate` to match against the last component of the file names (the "base name" of the files) you can use the '`--basename`' option. The opposite behaviour is the default, but can be selected explicitly by using the option '`--wholename`'.

The command

```
locate pattern
```

is almost equivalent to

```
find directories -name pattern
```

where *directories* are the directories for which the file name databases contain information. The differences are that the `locate` information might be out of date, and that `locate` handles wildcards in the pattern slightly differently than `find` (see Section 2.1.4 [Shell Pattern Matching], page 6).

The file name databases contain lists of files that were on the system when the databases were last updated. The system administrator can choose the file name of the default database, the frequency with which the databases are updated, and the directories for which they contain entries.

Here is how to select which file name databases `locate` searches. The default is system-dependent.

`--database=`*path*
`-d` *path*
: Instead of searching the default file name database, search the file name databases in *path*, which is a colon-separated list of database file names. (A semicolon is used on MS-DOS systems.) You can also use the environment variable `LOCATE_PATH` to set the list of database files to search. The option overrides the environment variable if both are used.

2.1.4 Shell Pattern Matching

`find` and `locate` can compare file names, or parts of file names, to shell patterns. A *shell pattern* is a string that may contain the following special characters, which are known as *wildcards* or *metacharacters*.

You must quote patterns that contain metacharacters to prevent the shell from expanding them itself. Double and single quotes both work; so does escaping with a backslash. (Don't use backslashes on MS-DOS, except for escaping quote characters, because backslashes in other places are treated as directory separators.)

`*`
: Matches any zero or more characters.

`?`
: Matches any one character.

`[`*string*`]`
: Matches exactly one character that is a member of the string *string*. This is called a *character class*. As a shorthand, *string* may contain ranges, which consist of two characters with a dash between them. For example, the class '`[a-z0-9_]`' matches a lowercase letter, a number, or an underscore. You can negate a class by placing a '`!`' or '`^`' immediately after the opening bracket. Thus, '`[^A-Z@]`' matches any character except an uppercase letter or an at sign.

\ Removes the special meaning of the character that follows it. This works even in character classes. (On MS-DOS, only the quote characters ! are protected by the backslash.)

In the `find` tests that do shell pattern matching ('`-name`', '`-wholename`', etc.), wildcards in the pattern will match a '`.`' at the beginning of a file name. This is also the case for `locate`. Thus, '`find -name '*macs'`' will match a file named `.emacs`, as will '`locate '*macs'`'.

Slash characters have no special significance in the shell pattern matching that `find` and `locate` do, unlike in the shell, in which wildcards do not match them. Therefore, a pattern '`foo*bar`' can match a file name '`foo3/bar`', and a pattern '`./sr*sc`' can match a file name '`./src/misc`'.

If you want to locate some files with the '`locate`' command but don't need to see the full list you can use the '`--limit`' option to see just a small number of results, or the '`--count`' option to display only the total number of matches.

2.2 Links

There are two ways that files can be linked together. *Symbolic links* are a special type of file whose contents are a portion of the name of another file. *Hard links* are multiple directory entries for one file; the file names all have the same index node (*inode*) number on the disk.

2.2.1 Symbolic Links

Symbolic links are names that reference other files. GNU `find` will handle symbolic links in one of two ways; firstly, it can dereference the links for you - this means that if it comes across a symbolic link, it examines the file that the link points to, in order to see if it matches the criteria you have specified. Secondly, it can check the link itself in case you might be looking for the actual link. If the file that the symbolic link points to is also within the directory hierarchy you are searching with the `find` command, you may not see a great deal of difference between these two alternatives.

By default, `find` examines symbolic links themselves when it finds them (and, if it later comes across the linked-to file, it will examine that, too). If you would prefer `find` to dereference the links and examine the file that each link points to, specify the '`-L`' option to `find`. You can explicitly specify the default behaviour by using the '`-P`' option. The '`-H`' option is a half-way-between option which ensures that any symbolic links listed on the command line are dereferenced, but other symbolic links are not.

Symbolic links are different to "hard links" in the sense that you need permissions upon the linked-to file in order to be able to dereference the link. This can mean that even if you specify the '`-L`' option, `find` may not be able to determine the properties of the file that the link points to (because you don't have sufficient permissions). In this situation, `find` uses the properties of the link itself. This also occurs if a symbolic link exists but points to a file that is missing.

The options controlling the behaviour of `find` with respect to links are as follows :-

'`-P`' `find` does not dereference symbolic links at all. This is the default behaviour. This option must be specified before any of the file names on the command line.

'-H' `find` does not dereference symbolic links (except in the case of file names on the command line, which are dereferenced). If a symbolic link cannot be dereferenced, the information for the symbolic link itself is used. This option must be specified before any of the file names on the command line.

'-L' `find` dereferences symbolic links where possible, and where this is not possible it uses the properties of the symbolic link itself. This option must be specified before any of the file names on the command line. Use of this option also implies the same behaviour as the '-noleaf' option. If you later use the '-H' or '-P' options, this does not turn off '-noleaf'.

'-follow' This option forms part of the "expression" and must be specified after the file names, but it is otherwise equivalent to '-L'. The '-follow' option affects only those tests which appear after it on the command line. This option is deprecated. Where possible, you should use '-L' instead.

The following differences in behavior occur when the '-L' option is used:

- `find` follows symbolic links to directories when searching directory trees.
- '-lname' and '-ilname' always return false (unless they happen to match broken symbolic links).
- '-type' reports the types of the files that symbolic links point to. This means that in combination with '-L', '-type l' will be true only for broken symbolic links. To check for symbolic links when '-L' has been specified, use '-xtype l'.
- Implies '-noleaf' (see Section 2.9 [Directories], page 14).

If the '-L' option or the '-H' option is used, the file names used as arguments to '-newer', '-anewer', and '-cnewer' are dereferenced and the timestamp from the pointed-to file is used instead (if possible – otherwise the timestamp from the symbolic link is used).

-lname *pattern* [Test]
-ilname *pattern* [Test]

True if the file is a symbolic link whose contents match shell pattern *pattern*. For '-ilname', the match is case-insensitive. See Section 2.1.4 [Shell Pattern Matching], page 6, for details about the *pattern* argument. If the '-L' option is in effect, this test will always return false for symbolic links unless they are broken. So, to list any symbolic links to `sysdep.c` in the current directory and its subdirectories, you can do:

```
find . -lname '*sysdep.c'
```

2.2.2 Hard Links

Hard links allow more than one name to refer to the same file. To find all the names which refer to the same file as NAME, use '-samefile NAME'. If you are not using the '-L' option, you can confine your search to one filesystem using the '-xdev' option. This is useful because hard links cannot point outside a single filesystem, so this can cut down on needless searching.

If the '-L' option is in effect, and NAME is in fact a symbolic link, the symbolic link will be dereferenced. Hence you are searching for other links (hard or symbolic) to the file pointed to by NAME. If '-L' is in effect but NAME is not itself a symbolic link, other symbolic links to the file NAME will be matched.

You can also search for files by inode number. This can occasionally be useful in diagnosing problems with filesystems for example, because `fsck` tends to print inode numbers. Inode numbers also occasionally turn up in log messages for some types of software, and are used to support the `ftok()` library function.

You can learn a file's inode number and the number of links to it by running '`ls -li`' or '`find -ls`'.

You can search for hard links to inode number NUM by using '`-inum NUM`'. If there are any filesystem mount points below the directory where you are starting the search, use the '`-xdev`' option unless you are also using the '`-L`' option. Using '`-xdev`' this saves needless searching, since hard links to a file must be on the same filesystem. See Section 2.10 [Filesystems], page 16.

-samefile *NAME* [Test]

File is a hard link to the same inode as NAME. If the '`-L`' option is in effect, symbolic links to the same file as NAME points to are also matched.

-inum *n* [Test]

File has inode number *n*. The '`+`' and '`-`' qualifiers also work, though these are rarely useful. Much of the time it is easier to use '`-samefile`' rather than this option.

You can also search for files that have a certain number of links, with '`-links`'. Directories normally have at least two hard links; their `.` entry is the second one. If they have subdirectories, each of those also has a hard link called `..` to its parent directory. The `.` and `..` directory entries are not normally searched unless they are mentioned on the `find` command line.

-links *n* [Test]

File has *n* hard links.

-links +*n* [Test]

File has more than *n* hard links.

-links -*n* [Test]

File has fewer than *n* hard links.

2.3 Time

Each file has three time stamps, which record the last time that certain operations were performed on the file:

1. access (read the file's contents)
2. change the status (modify the file or its attributes)
3. modify (change the file's contents)

There is no timestamp that indicates when a file was *created*.

You can search for files whose time stamps are within a certain age range, or compare them to other time stamps.

2.3.1 Age Ranges

These tests are mainly useful with ranges ('`+n`' and '`-n`').

`-atime` *n* [Test]
`-ctime` *n* [Test]
`-mtime` *n* [Test]

True if the file was last accessed (or its status changed, or it was modified) *n**24 hours ago. The number of 24-hour periods since the file's timestamp is always rounded down; therefore 0 means "less than 24 hours ago", 1 means "between 24 and 48 hours ago", and so forth.

`-amin` *n* [Test]
`-cmin` *n* [Test]
`-mmin` *n* [Test]

True if the file was last accessed (or its status changed, or it was modified) *n* minutes ago. These tests provide finer granularity of measurement than '`-atime`' et al., but rounding is done in a similar way. For example, to list files in `/u/bill` that were last read from 2 to 6 minutes ago:

```
find /u/bill -amin +2 -amin -6
```

`-daystart` [Option]

Measure times from the beginning of today rather than from 24 hours ago. So, to list the regular files in your home directory that were modified yesterday, do

```
find ~ -daystart -type f -mtime 1
```

The '`-daystart`' option is unlike most other options in that it has an effect on the way that other tests are performed. The affected tests are '`-amin`', '`-cmin`', '`-mmin`', '`-atime`', '`-ctime`' and '`-mtime`'. The '`-daystart`' option only affects the behaviour of any tests which appear after it on the command line.

2.3.2 Comparing Timestamps

As an alternative to comparing timestamps to the current time, you can compare them to another file's timestamp. That file's timestamp could be updated by another program when some event occurs. Or you could set it to a particular fixed date using the `touch` command. For example, to list files in `/usr` modified after February 1 of the current year:

```
touch -t 02010000 /tmp/stamp$$
find /usr -newer /tmp/stamp$$
rm -f /tmp/stamp$$
```

`-anewer` *file* [Test]
`-cnewer` *file* [Test]
`-newer` *file* [Test]

True if the file was last accessed (or its status changed, or it was modified) more recently than *file* was modified. These tests are affected by '`-follow`' only if '`-follow`' comes before them on the command line. See Section 2.2.1 [Symbolic Links], page 7, for more information on '`-follow`'. As an example, to list any files modified since `/bin/sh` was last modified:

```
find . -newer /bin/sh
```

-used *n* [Test]

True if the file was last accessed *n* days after its status was last changed. Useful for finding files that are not being used, and could perhaps be archived or removed to save disk space.

2.4 Size

-size *n*[*bckwMG*] [Test]

True if the file uses *n* units of space, rounding up. The units are 512-byte blocks by default, but they can be changed by adding a one-character suffix to *n*:

`b` 512-byte blocks (never 1024)

`c` bytes

`k` kilobytes (1024 bytes)

`w` 2-byte words

`M` Megabytes (units of 1048576 bytes)

`G` Gigabytes (units of 1073741824 bytes)

The 'b' suffix always considers blocks to be 512 bytes. This is not affected by the setting (or non-setting) of the POSIXLY_CORRECT environment variable. This behaviour is different to the behaviour of the '`-ls`' action). If you want to use 1024-byte units, use the 'k' suffix instead.

The number can be prefixed with a '`+`' or a '`-`'. A plus sign indicates that the test should succeed if the file uses at least *n* units of storage (a common use of this test) and a minus sign indicates that the test should succeed if the file uses less than *n* units of storage. There is no '=' prefix, because that's the default anyway.

The size does not count indirect blocks, but it does count blocks in sparse files that are not actually allocated. In other words, it's consistent with the result you get for '`ls -l`' or '`wc -c`'. This handling of sparse files differs from the output of the '`%k`' and '`%b`' format specifiers for the '`-printf`' predicate.

-empty [Test]

True if the file is empty and is either a regular file or a directory. This might help determine good candidates for deletion. This test is useful with '`-depth`' (see Section 2.9 [Directories], page 14) and '`-delete`' (see Section 3.3.1 [Single File], page 23).

2.5 Type

-type *c* [Test]

True if the file is of type *c*:

`b` block (buffered) special

`c` character (unbuffered) special

`d` directory

`p` named pipe (FIFO)

`f` regular file

`l` symbolic link; if '`-L`' is in effect, this is true only for broken symbolic links. If you want to search for symbolic links when '`-L`' is in effect, use '`-xtype`' instead of '`-type`'.

`s` socket

`D` door (Solaris)

-xtype *c* [Test]

This test behaves the same as '`-type`' unless the file is a symbolic link. If the file is a symbolic link, the result is as follows (in the table below, '`X`' should be understood to represent any letter except '`l`'):

``-P -xtype l''
: True if the symbolic link is broken

``-P -xtype X''
: True if the (ultimate) target file is of type '`X`'.

``-L -xtype l''
: Always true

``-L -xtype X''
: False unless the symbolic link is broken

In other words, for symbolic links, '`-xtype`' checks the type of the file that '`-type`' does not check.

The '`-H`' option also affects the behaviour of '`-xtype`'. When '`-H`' is in effect, '`-xtype`' behaves as if '`-L`' had been specified when examining files listed on the command line, and as if '`-P`' had been specified otherwise. If neither '`-H`' nor '`-L`' was specified, '`-xtype`' behaves as if '`-P`' had been specified.

See Section 2.2.1 [Symbolic Links], page 7, for more information on '`-follow`' and '`-L`'.

2.6 Owner

-user *uname* [Test]
-group *gname* [Test]

True if the file is owned by user *uname* (belongs to group *gname*). A numeric ID is allowed.

-uid *n* [Test]
-gid *n* [Test]

True if the file's numeric user ID (group ID) is *n*. These tests support ranges ('`+n`' and '`-n`'), unlike '`-user`' and '`-group`'.

-nouser [Test]
-nogroup [Test]

True if no user corresponds to the file's numeric user ID (no group corresponds to the numeric group ID). These cases usually mean that the files belonged to users who have since been removed from the system. You probably should change the ownership of such files to an existing user or group, using the `chown` or `chgrp` program.

2.7 Permissions

See Chapter 5 [File Permissions], page 36, for information on how file permissions are structured and how to specify them.

`-perm` *mode* [Test]

True if the file's permissions are exactly *mode*, which can be numeric or symbolic.

If *mode* starts with '`-`', true if *all* of the permissions set in *mode* are set for the file; permissions not set in *mode* are ignored.

If *mode* starts with '`/`', true if *any* of the permissions set in *mode* are set for the file; permissions not set in *mode* are ignored. This is a GNU extension.

If you don't use the '`/`' or '`-`' form with a symbolic mode string, you may have to specify a rather complex mode string. For example '`-perm g=w`' will only match files which have mode 0020 (that is, ones for which group write permission is the only permission set). It is more likely that you will want to use the '`/`' or '`-`' forms, for example '`-perm -g=w`', which matches any file with group write permission.

'`-perm 664`'
: Match files which have read and write permission for their owner, and group, but which the rest of the world can read but not write to. Files which meet these criteria but have other permissions bits set (for example if someone can execute the file) will not be matched.

'`-perm -664`'
: Match files which have read and write permission for their owner, and group, but which the rest of the world can read but not write to, without regard to the presence of any extra permission bits (for example the executable bit). This will match a file which has mode 0777, for example.

'`-perm /222`'
: Match files which are writable by somebody (their owner, or their group, or anybody else).

'`-perm /022`'
: Match files which are writable by either their owner or their group. The files don't have to be writable by both the owner and group to be matched; either will do.

'`-perm /g+w,o+w`'
: As above.

'`-perm /g=w,o=w`'
: As above

'`-perm -022`'
: Search for files which are writable by both their owner and their group.

'-perm -444 -perm /222 ! -perm /111'
: Search for files which are readable for everybody, have at least one write bit set (i.e. somebody can write to them), but which cannot be executed by anybody. Note that in some shells the '!' must be escaped;.

'-perm -a+r -perm /a+w ! -perm /a+x'
: As above.

'-perm -g+w,o+w'
: As above.

> **Warning:** If you specify '-perm /000' or '-perm /mode' where the symbolic mode 'mode' has no bits set, the test currently matches no files. This differs from the behaviour of '-perm -000', which matches all files. The behaviour of '-perm /000' will be changed to be consistent with the behaviour of '-perm -000'. The change will probably be made in early 2006.

2.8 Contents

To search for files based on their contents, you can use the `grep` program. For example, to find out which C source files in the current directory contain the string 'thing', you can do:

```
grep -l thing *.[ch]
```

If you also want to search for the string in files in subdirectories, you can combine `grep` with `find` and `xargs`, like this:

```
find . -name '*.[ch]' | xargs grep -l thing
```

The '-l' option causes `grep` to print only the names of files that contain the string, rather than the lines that contain it. The string argument ('thing') is actually a regular expression, so it can contain metacharacters. This method can be refined a little by using the '-r' option to make `xargs` not run `grep` if `find` produces no output, and using the `find` action '-print0' and the `xargs` option '-0' to avoid misinterpreting files whose names contain spaces:

```
find . -name '*.[ch]' -print0 | xargs -r -0 grep -l thing
```

For a fuller treatment of finding files whose contents match a pattern, see the manual page for `grep`.

2.9 Directories

Here is how to control which directories `find` searches, and how it searches them. These two options allow you to process a horizontal slice of a directory tree.

`-maxdepth` *levels* [Option]
: Descend at most *levels* (a non-negative integer) levels of directories below the command line arguments. '-maxdepth 0' means only apply the tests and actions to the command line arguments.

`-mindepth` *levels* [Option]
: Do not apply any tests or actions at levels less than *levels* (a non-negative integer). '-mindepth 1' means process all files except the command line arguments.

`-depth` [Option]

Process each directory's contents before the directory itself. Doing this is a good idea when producing lists of files to archive with `cpio` or `tar`. If a directory does not have write permission for its owner, its contents can still be restored from the archive since the directory's permissions are restored after its contents.

`-d` [Option]

This is a deprecated synonym for '`-depth`', for compatibility with Mac OS X, FreeBSD and OpenBSD. The '`-depth`' option is a POSIX feature, so it is better to use that.

`-prune` [Action]

If the file is a directory, do not descend into it. The result is true. For example, to skip the directory `src/emacs` and all files and directories under it, and print the names of the other files found:

```
find . -wholename './src/emacs' -prune -o -print
```

The above command will not print `./src/emacs` among its list of results. This however is not due to the effect of the '`-prune`' action (which only prevents further descent, it doesn't make sure we ignore that item). Instead, this effect is due to the use of '`-o`'. Since the left hand side of the "or" condition has succeeded for `./src/emacs`, it is not necessary to evaluate the right-hand-side ('`-print`') at all for this particular file. If you wanted to print that directory name you could use either an extra '`-print`' action:

```
find . -wholename './src/emacs' -prune -print -o -print
```

or use the comma operator:

```
find . -wholename './src/emacs' -prune , -print
```

If the '`-depth`' option is in effect, the subdirectories will have already been visited in any case. Hence '`-prune`' has no effect and returns false.

`-quit` [Action]

Exit immediately (with return value zero if no errors have occurred). This is different to '`-prune`' because '`-prune`' only applies to the contents of pruned directories, whilt '`-quit`' simply makes `find` stop immediately. No child processes will be left running, but no more files specified on the command line will be processed. For example, `find /tmp/foo /tmp/bar -print -quit` will print only '`/tmp/foo`'. Any command lines which have been built by '`-exec ... \+`' or '`-execdir ... \+`' are invoked before the program is exited.

`-noleaf` [Option]

Do not optimize by assuming that directories contain 2 fewer subdirectories than their hard link count. This option might be needed when searching filesystems that do not follow the Unix directory-link convention, such as CD-ROM or MS-DOS filesystems or AFS volume mount points. Each directory on a normal Unix filesystem has at least 2 hard links: its name and its `.` entry. Additionally, its subdirectories (if any) each have a `..` entry linked to that directory. When `find` is examining a directory, after it has statted 2 fewer subdirectories than the directory's link count, it knows that the rest of the entries in the directory are non-directories (*leaf* files in the directory tree). If only the files' names need to be examined, there is no need to stat them; this gives a significant increase in search speed. Usually, you won't need this option even on MS-DOS systems, so unless you see that some subdirectories aren't listed, don't use it: it makes `find` run 2 to 4 times slower.

`-ignore_readdir_race` [Option]

If a file disappears after its name has been read from a directory but before `find` gets around to examining the file with `stat`, don't issue an error message. If you don't specify this option, an error message will be issued. This option can be useful in system scripts (cron scripts, for example) that examine areas of the filesystem that change frequently (mail queues, temporary directories, and so forth), because this scenario is common for those sorts of directories. Completely silencing error messages from `find` is undesirable, so this option neatly solves the problem. There is no way to search one part of the filesystem with this option on and part of it with this option off, though. When this option is turned on and find discovers that one of the start-point files specified on the command line does not exist, no error message will be issued.

`-noignore_readdir_race` [Option]

This option reverses the effect of the '`-ignore_readdir_race`' option.

2.10 Filesystems

A *filesystem* is a section of a disk, either on the local host or mounted from a remote host over a network. Searching network filesystems can be slow, so it is common to make `find` avoid them.

There are two ways to avoid searching certain filesystems. One way is to tell `find` to only search one filesystem:

`-xdev` [Option]
`-mount` [Option]

Don't descend directories on other filesystems. These options are synonyms.

The other way is to check the type of filesystem each file is on, and not descend directories that are on undesirable filesystem types:

`-fstype` *type* [Test]

True if the file is on a filesystem of type *type*. The valid filesystem types vary among different versions of Unix; an incomplete list of filesystem types that are accepted on some version of Unix or another is:

```
ext2 ext3 proc sysfs ufs 4.2 4.3 nfs tmp mfs S51K S52K
```

The DJGPP port of `find` recognizes the following filesystem types:

`hd` a hard (aka fixed) disk

`fd` a floppy disk

`cdrom` a CD-ROM disk

`ram` a RAM-disk

`net` a networked drive

`dblsp` a disk compressed with the DblSpace method `stac` a disk compressed with the Stacker method

`subst` a SUBSTed disk

`join` a JOINed disk

You can use '`-printf`' with the '`%F`' directive to see the types of your filesystems. The '`%D`' directive shows the device number. See Section 3.2 [Print File Information], page 18. '`-fstype`' is usually used with '`-prune`' to avoid searching remote filesystems or removable media (see Section 2.9 [Directories], page 14).

2.11 Combining Primaries With Operators

Operators build a complex expression from tests and actions. The operators are, in order of decreasing precedence:

`( expr )` Force precedence. True if *expr* is true.

`! expr`
`-not expr` True if *expr* is false. In some shells, it is necessary to protect the '`!`' from shell interpretation by quoting it.

`expr1 expr2`
`expr1 -a expr2`
`expr1 -and expr2`
And; *expr2* is not evaluated if *expr1* is false.

`expr1 -o expr2`
`expr1 -or expr2`
Or; *expr2* is not evaluated if *expr1* is true.

`expr1 , expr2`
List; both *expr1* and *expr2* are always evaluated. True if *expr2* is true. The value of *expr1* is discarded. This operator lets you do multiple independent operations on one traversal, without depending on whether other operations succeeded. The two operations *expr1* and *expr2* are not always fully independent, since *expr1* might have side effects like touching or deleting files, or it might use '`-prune`' which would also affect *expr2*.

`find` searches the directory tree rooted at each file name by evaluating the expression from left to right, according to the rules of precedence, until the outcome is known (the left hand side is false for '`-and`', true for '`-or`'), at which point `find` moves on to the next file name.

There are two other tests that can be useful in complex expressions:

`-true` [Test]
Always true.

`-false` [Test]
Always false.

3 Actions

There are several ways you can print information about the files that match the criteria you gave in the `find` expression. You can print the information either to the standard output or to a file that you name. You can also execute commands that have the file names as arguments. You can use those commands as further filters to select files.

3.1 Print File Name

`-print` [Action]

True; print the entire file name on the standard output, followed by a newline. If there is the faintest possibility that one of the files for which you are searching might contain a newline, you should use '`-print0`' instead.

`-fprint` *file* [Action]

True; print the entire file name into file *file*, followed by a newline. If *file* does not exist when `find` is run, it is created; if it does exist, it is truncated to 0 bytes. The named output file is always created, even if no output is sent to it. The file names `/dev/stdout` and `/dev/stderr` are handled specially; they refer to the standard output and standard error output, respectively.

If there is the faintest possibility that one of the files for which you are searching might contain a newline, you should use '`-fprint0`' instead.

3.2 Print File Information

`-ls` [Action]

True; list the current file in '`ls -dils`' format on the standard output. The output looks like this:

```
204744   17 -rw-r--r--   1 djm      staff       17337 Nov  2  1992 ./lwall-quotes
```

The fields are:

1. The inode number of the file. See Section 2.2.2 [Hard Links], page 8, for how to find files based on their inode number.
2. the number of blocks in the file. The block counts are of 1K blocks, unless the environment variable `POSIXLY_CORRECT` is set, in which case 512-byte blocks are used. See Section 2.4 [Size], page 11, for how to find files based on their size.
3. The file's type and permissions. The type is shown as a dash for a regular file; for other file types, a letter like for '`-type`' is used (see Section 2.5 [Type], page 11). The permissions are read, write, and execute for the file's owner, its group, and other users, respectively; a dash means the permission is not granted. See Chapter 5 [File Permissions], page 36, for more details about file permissions. See Section 2.7 [Permissions], page 13, for how to find files based on their permissions.
4. The number of hard links to the file.
5. The user who owns the file.
6. The file's group.
7. The file's size in bytes.

8. The date the file was last modified.
9. The file's name. '-ls' quotes non-printable characters in the file names using C-like backslash escapes. This may change soon, as the treatment of unprintable characters is harmonised for '-ls', '-fls', '-print', '-fprint', '-printf' and '-fprintf'.

-fls *file* [Action]

True; like '-ls' but write to *file* like '-fprint' (see Section 3.1 [Print File Name], page 18). The named output file is always created, even if no output is sent to it.

-printf *format* [Action]

True; print *format* on the standard output, interpreting '\' escapes and '%' directives. Field widths and precisions can be specified as with the `printf` C function. Format flags (like '#' for example) may not work as you expect because many of the fields, even numeric ones, are printed with %s. Numeric flags which are affected in this way include G, U, b, D, k and n. This difference in behaviour means though that the format flag '-' will work; it forces left-alignment of the field. Unlike '-print', '-printf' does not add a newline at the end of the string. If you want a newline at the end of the string, add a '\n'.

-fprintf *file format* [Action]

True; like '-printf' but write to *file* like '-fprint' (see Section 3.1 [Print File Name], page 18). The output file is always created, even if no output is ever sent to it.

When you work under `COMMAND.COM` or any of its work-alikes, you have to double every '%' character, because they are special to those shells.

3.2.1 Escapes

The escapes that '-printf' and '-fprintf' recognise are:

`\a` Alarm bell.

`\b` Backspace.

`\c` Stop printing from this format immediately and flush the output.

`\f` Form feed.

`\n` Newline.

`\r` Carriage return.

`\t` Horizontal tab.

`\v` Vertical tab.

`\\` A literal backslash ('\').

`\0` ASCII NUL.

`\NNN` The character whose ASCII code is NNN (octal).

A '\' character followed by any other character is treated as an ordinary character, so they both are printed, and a warning message is printed to the standard error output (because it was probably a typo).

3.2.2 Format Directives

'-printf' and '-fprintf' support the following format directives to print information about the file being processed. The C printf function, field width and precision specifiers are supported, as applied to string (%s) types. That is, you can specify "minimum field width"."maximum field width" for each directive. Format flags (like '#' for example) may not work as you expect because many of the fields, even numeric ones, are printed with %s. The format flag '-' does work; it forces left-alignment of the field.

On MS-DOS, you will have to type 2 '%' characters for each one you want to pass to find. That's because one of them is removed by the standard shell COMMAND.COM which uses '%' for environment variable substitution. Thus, you will have to write '%%f' for the filename and '%%%%' for the literal '%'.

'%%' is a literal percent sign. A '%' character followed by an unrecognised character (i.e., not a known directive or printf field width and precision specifier), is discarded (but the unrecognised character is printed), and a warning message is printed to the standard error output (because it was probably a typo). Don't rely on this behaviour, because other directives may be added in the future.

A '%' at the end of the format argument causes undefined behaviour since there is no following character. In some locales, it may hide your door keys, while in others it may remove the final page from the novel you are reading.

3.2.2.1 Name Directives

%p
: File's name (not the absolute path name, but the name of the file as it was encountered by find - that is, as a relative path from one of the starting points).

%f
: File's name with any leading directories removed (only the last element).

%h
: Leading directories of file's name (all but the last element and the slash before it). If the file's name contains no slashes (for example because it was named on the command line and is in the current working directory), then "%h" expands to ".". This prevents "%h/%f" expanding to "/foo", which would be surprising and probably not desirable.

%P
: File's name with the name of the command line argument under which it was found removed from the beginning.

%H
: Command line argument under which file was found.

3.2.2.2 Ownership Directives

%g
: File's group name, or numeric group ID if the group has no name.

%G
: File's numeric group ID.

%u
: File's user name, or numeric user ID if the user has no name.

%U
: File's numeric user ID.

%m
: File's permissions (in octal). If you always want to have a leading zero on the number, use the '#' format flag, for example '%#m'.

  The permission numbers used are the traditional Unix permission numbers, which will be as expected on most systems, but if your system's permission bit layout differs from the traditional Unix semantics, you will see a difference between the mode as printed by '%m' and the mode as it appears in `struct stat`.

%M File's permissions (in symbolic form, as for `ls`). This directive is supported in findutils 4.2.5 and later.

3.2.2.3 Size Directives

%k The amount of disk space used for this file in 1K blocks. Since disk space is allocated in multiples of the filesystem block size this is usually greater than %s/1024, but it can also be smaller if the file is a sparse file (that is, it has "holes").

%b The amount of disk space used for this file in 512-byte blocks. Since disk space is allocated in multiples of the filesystem block size this is usually greater than %s/512, but it can also be smaller if the file is a sparse file (that is, it has "holes").

%s File's size in bytes.

3.2.2.4 Location Directives

%d File's depth in the directory tree (depth below a file named on the command line, not depth below the root directory). Files named on the command line have a depth of 0. Subdirectories immediately below them have a depth of 1, and so on.

%D The device number on which the file exists (the `st_dev` field of `struct stat`), in decimal.

%F Type of the filesystem the file is on; this value can be used for '-fstype' (see Section 2.9 [Directories], page 14).

%l Object of symbolic link (empty string if file is not a symbolic link).

%i File's inode number (in decimal).

%n Number of hard links to file.

%y Type of the file as used with '-type'. If the file is a symbolic link, 'l' will be printed.

%Y Type of the file as used with '-type'. If the file is a symbolic link, it is dereferenced. If the file is a broken symbolic link, 'N' is printed.

3.2.2.5 Time Directives

Some of these directives use the C `ctime` function. Its output depends on the current locale, but it typically looks like

```
Wed Nov  2 00:42:36 1994
```

%a File's last access time in the format returned by the C `ctime` function.

%A*k* File's last access time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 22).

`%c` File's last status change time in the format returned by the C `ctime` function.

`%Ck` File's last status change time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 22).

`%t` File's last modification time in the format returned by the C `ctime` function.

`%Tk` File's last modification time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 22).

3.2.3 Time Formats

Below are the formats for the directives '`%A`', '`%C`', and '`%T`', which print the file's timestamps. Some of these formats might not be available on all systems, due to differences in the C `strftime` function between systems.

3.2.3.1 Time Components

The following format directives print single components of the time.

`H` hour (00..23)

`I` hour (01..12)

`k` hour (0..23)

`l` hour (1..12)

`p` locale's AM or PM

`Z` time zone (e.g., EDT), or nothing if no time zone is determinable

`M` minute (00..59)

`S` second (00..61)

`@` seconds since Jan. 1, 1970, 00:00 GMT.

3.2.3.2 Date Components

The following format directives print single components of the date.

`a` locale's abbreviated weekday name (Sun..Sat)

`A` locale's full weekday name, variable length (Sunday..Saturday)

`b`
`h` locale's abbreviated month name (Jan..Dec)

`B` locale's full month name, variable length (January..December)

`m` month (01..12)

`d` day of month (01..31)

`w` day of week (0..6)

`j` day of year (001..366)

`U` week number of year with Sunday as first day of week (00..53)

`W` week number of year with Monday as first day of week (00..53)

`Y` year (1970. . .)

`y` last two digits of year (00..99)

3.2.3.3 Combined Time Formats

The following format directives print combinations of time and date components.

`r` time, 12-hour (hh:mm:ss [AP]M)

`T` time, 24-hour (hh:mm:ss)

`X` locale's time representation (H:M:S)

`c` locale's date and time (Sat Nov 04 12:02:33 EST 1989)

`D` date (mm/dd/yy)

`x` locale's date representation (mm/dd/yy)

`+` Date and time, separated by '+', for example '2004-04-28+22:22:05'. The time is given in the current timezone (which may be affected by setting the TZ environment variable). This is a GNU extension.

3.2.3.4 Formatting Flags

The '`%m`' and '`%d`' directives support the '`#`', '`0`' and '`+`' flags, but the other directives do not, even if they print numbers. Numeric directives that do not support these flags include

'`G`', '`U`', '`b`', '`D`', '`k`' and '`n`'.

All fields support the format flag '`-`', which makes fields left-aligned. That is, if the field width is greater than the actual contents of the field, the requisite number of spaces are printed after the field content instead of before it.

3.3 Run Commands

You can use the list of file names created by `find` or `locate` as arguments to other commands. In this way you can perform arbitrary actions on the files.

3.3.1 Single File

Here is how to run a command on one file at a time.

`-execdir` *command ;* [Action]

Execute *command*; true if zero status is returned. `find` takes all arguments after '`-exec`' to be part of the command until an argument consisting of '`;`' is reached. It replaces the string '`{}`' by the current file name being processed everywhere it occurs in the command. Both of these constructions need to be escaped (with a '\') or quoted to protect them from expansion by the shell. The command is executed in the directory in which `find` was run.

For example, to compare each C header file in or below the current directory with the file `/tmp/master`:

```
find . -name '*.h' -execdir diff -u '{}' /tmp/master ';'
```

If you use''`-execdir`', you must ensure that the current directory is not on *$PATH*, because otherwise an attacker could make '`find`' run commands of their choice simply by leaving a suitably-named file in the right directory. GNU find will refuse to run if you use '`-execdir`' and the current directory is in *$PATH*.

Another similar option, '`-exec`' is supported, but is less secure. See Chapter 9 [Security Considerations], page 70, for a discussion of the security problems surrounding '`-exec`'.

`-exec` *command ;* [Action]

This insecure variant of the '`-execdir`' action is specified by POSIX. The main difference is that the command is executed in the directory from which `find` was invoked, meaning that '`{}`' is expanded to a relative path starting with the name of one of the starting directories, rather than just the basename of the matched file.

While some implementations of `find` replace the '`{}`' only where it appears on its own in an argument, GNU `find` replaces '`{}`' wherever it appears.

`-dosexec` *command ;* [Action]

Like '`-exec`', but submits the filename to *command* in MS-DOS format by converting all Unix-style forward slashes to DOS-style backslashes. This is useful for invoking native DOS programs and commands internal to `COMMAND.COM` that don't understand Unix-style forward slashes. Note that the above conversion is done unconditionally and you cannot avoid it by using any quotes; in particular, `find` couldn't care less if the arguments aren't filenames at all. Also note that the way this action is implemented, it affects all the '`-exec`' and '`-ok`' (see Section 3.3.3 [Querying], page 30) actions for this invocation of `find`; so mixing '`-exec`', '`-ok`' and '`-dosexec`' might be unwise unless the programs involved understand both kinds of slashes.

For example, here is a command that will rename each file with a `.cpp` extension to have a `.cc` extension, in the current directory and all of its subdirectories:

```
find . -name '*.cpp' -dosexec command /c ren {} '*.cc' ;
```

(Note that on MS-DOS you don't have to quote the '`{}`' construct and the semicolon.)

3.3.2 Multiple Files

Sometimes you need to process files one at a time. But usually this is not necessary, and, it is faster to run a command on as many files as possible at a time, rather than once per file. Doing this saves on the time it takes to start up the command each time.

The '`-execdir`' and '`-exec`' actions have variants that build command lines containing as many matched files as possible.

`-execdir` *command {}* `+` [Action]

This works as for '`-execdir command ;`', except that the '`{}`' at the end of the command is expanded to a list of names of matching files. This expansion is done in such a way as to avoid exceeding the maximum command line length available on the system. Only one '`{}`' is allowed within the command, and it must appear at the end, immediately before the '`+`'. A '`+`' appearing in any position other than immediately after '`{}`' is not considered to be special (that is, it does not terminate the command).

`-exec` *command {}* `+` [Action]

This insecure variant of the '`-execdir`' action is specified by POSIX. The main difference is that the command is executed in the directory from which `find` was invoked, meaning that '`{}`' is expanded to a relative path starting with the name of one of the starting directories, rather than just the basename of the matched file.

Before `find` exits, any partially-built command lines are executed. This happens even if the exit was caused by the '`-quit`' action. However, some types of error (for example not being able to invoke `stat()` on the current directory) can cause an immediate fatal exit. In this situation, any partially-built command lines will not be invoked (this prevents possible infinite loops).

At first sight, it looks like the list of filenames to be processed can only be at the end of the command line, and that this might be a problem for some comamnds (`cp` and `rsync` for example).

However, there is a slightly obscure but powerful workarouund for this problem which takes advantage of the behaviour of `sh -c`:-

```
find startpoint -tests ... -exec sh -c 'scp "$@" remote:/dest' sh {} +
```

In the example above, the filenames we want to work on need to occur on the `scp` command line before the name of the destination. We use the shell to invoke the command `scp "$@" remote:/dest` and the shell expands `"$@"` to the list of filenames we want to process.

Another, but less secure, way to run a command on more than one file at once, is to use the `xargs` command, which is invoked like this:

```
xargs [option...] [command [initial-arguments]]
```

`xargs` normally reads arguments from the standard input. These arguments are delimited by blanks (which can be protected with double or single quotes or a backslash) or newlines. It executes the *command* (default is `echo` on MS-DOS systems and `/bin/echo` elsewhere) one or more times with any *initial-arguments* followed by arguments read from standard input. Blank lines on the standard input are ignored.

Instead of blank-delimited names, it is safer to use '`find -print0`' or '`find -fprint0`' and process the output by giving the '`-0`' or '`--null`' option to GNU `xargs`, GNU `tar`, GNU `cpio`, or `perl`. The `locate` command also has a '`-0`' or '`--null`' option which does the same thing.

Except with MS-DOS native shells (that don't support command substitution), you can use shell command substitution (backquotes) to process a list of arguments, like this:

```
grep -l sprintf `find $HOME -name '*.c' -print`
```

However, that method produces an error if the length of the '`.c`' file names exceeds the operating system's command line length limit. `xargs` avoids that problem by running the command as many times as necessary without exceeding the limit:

```
find $HOME -name '*.c' -print | xargs grep -l sprintf
```

However, if the command needs to have its standard input be a terminal (`less`, for example), you have to use the shell command substitution method or use the '`--arg-file`' option of `xargs`.

The `xargs` command will process all its input, building command lines and executing them, unless one of the commands exits with a status of 255 (this will cause xargs to issue an error message and stop) or it reads a line contains the end of file string specified with the '`--eof`' option.

3.3.2.1 Unsafe File Name Handling

Because file names can contain quotes, backslashes, blank characters, and even newlines, it is not safe to process them using `xargs` in its default mode of operation. But since most files' names do not contain blanks, this problem occurs only infrequently. If you are only searching through files that you know have safe names, then you need not be concerned about it.

In many applications, if `xargs` botches processing a file because its name contains special characters, some data might be lost. The importance of this problem depends on the importance of the data and whether anyone notices the loss soon enough to correct it. However, here is an extreme example of the problems that using blank-delimited names can cause. If the following command is run daily from `cron`, then any user can remove any file on the system:

```
find / -name '#*' -atime +7 -print | xargs rm
```

For example, you could do something like this:

```
eg$ echo > '#
vmunix'
```

and then `cron` would delete `/vmunix`, if it ran `xargs` with `/` as its current directory.

To delete other files, for example `/u/joeuser/.plan`, you could do this:

```
eg$ mkdir '#
'
eg$ cd '#
'
eg$ mkdir u u/joeuser u/joeuser/.plan'
'
eg$ echo > u/joeuser/.plan'
/#foo'
eg$ cd ..
eg$ find . -name '#*' -print | xargs echo
./# ./# /u/joeuser/.plan /#foo
```

3.3.2.2 Safe File Name Handling

Here is how to make `find` output file names so that they can be used by other programs without being mangled or misinterpreted. You can process file names generated this way by giving the '`-0`' or '`--null`' option to GNU `xargs`, GNU `tar`, GNU `cpio`, or `perl`.

`-print0` [Action]

True; print the entire file name on the standard output, followed by a null character.

`-fprint0` *file* [Action]

True; like '`-print0`' but write to *file* like '`-fprint`' (see Section 3.1 [Print File Name], page 18). The output file is always created.

As of findutils version 4.2.4, the `locate` program also has a '`--null`' option which does the same thing. For similarity with `xargs`, the short form of the option '`-0`' can also be used.

If you want to be able to handle file names safely but need to run commands which want to be connected to a terminal on their input, you can use the '`--arg-file`' option to `xargs` like this:

```
find / -name xyzzy -print0 > list
xargs --null --arg-file=list munge
```

The example above runs the `munge` program on all the files named `xyzzy` that we can find, but `munge`'s input will still be the terminal (or whatever the shell was using as standard input). If your shell has the "process substitution" feature '`<(...)`', you can do this in just one step:

```
xargs --null --arg-file=<(find / -name xyzzy -print0) munge
```

3.3.2.3 Unusual Characters in File Names

As discussed above, you often need to be careful about how the names of files are handled by `find` and other programs. If the output of `find` is not going to another program but instead is being shown on a terminal, this can still be a problem. For example, some character sequences can reprogram the function keys on some terminals. See Chapter 9 [Security Considerations], page 70, for a discussion of other security problems relating to `find`.

Unusual characters are handled differently by various actions, as described below.

'`-print0`'
'`-fprint0`'
: Always print the exact file name, unchanged, even if the output is going to a terminal.

'`-ok`'
'`-okdir`'
: Always print the exact file name, unchanged. This will probably change in a future release.

'`-ls`'
'`-fls`'
: Unusual characters are always escaped. White space, backslash, and double quote characters are printed using C-style escaping (for example '`\f`', '`\"`'). Other unusual characters are printed using an octal escape. Other printable characters (for '`-ls`' and '`-fls`' these are the characters between octal 041 and 0176) are printed as-is.

'`-printf`'
'`-fprintf`'
: If the output is not going to a terminal, it is printed as-is. Otherwise, the result depends on which directive is in use:

  %D, %F, %H, %Y, %y
  : These expand to values which are not under control of files' owners, and so are printed as-is.

  %a, %b, %c, %d, %g, %G, %i, %k, %m, %M, %n, %s, %t, %u, %U
  : These have values which are under the control of files' owners but which cannot be used to send arbitrary data to the terminal, and so these are printed as-is.

  %f, %h, %l, %p, %P
  : The output of these directives is quoted if the output is going to a terminal.

    This quoting is performed in the same way as for GNU `ls`. This is not the same quoting mechanism as the one used for '`-ls`' and 'fls'. If you are able to decide what format to use for the output of find then it is normally better to use '\0' as a terminator than to use newline, as file names can contain white space and newline characters.

'-print'
'-fprint' Quoting is handled in the same way as for the '%p' directive of '-printf' and '-fprintf'. If you are using find in a script or in a situation where the matched files might have arbitrary names, you should consider using '-print0' instead of '-print'.

The locate program quotes and escapes unusual characters in file names in the same way as find's '-print' action.

The behaviours described above may change soon, as the treatment of unprintable characters is harmonised for '-ls', '-fls', '-print', '-fprint', '-printf' and '-fprintf'.

3.3.2.4 Limiting Command Size

xargs gives you control over how many arguments it passes to the command each time it executes it. By default, it uses up to ARG_MAX - 2k, or 128k, whichever is smaller, characters per command. It uses as many lines and arguments as fit within that limit. The following options modify those values.

--no-run-if-empty
-r If the standard input does not contain any nonblanks, do not run the command. By default, the command is run once even if there is no input. This option is a GNU extension.

--max-lines[=*max-lines*]
-L *max-lines*
-l[*max-lines*]
Use at most *max-lines* nonblank input lines per command line; *max-lines* defaults to 1 if omitted; omitting the argument is not allowed in the case of the '-L' option. Trailing blanks cause an input line to be logically continued on the next input line, for the purpose of counting the lines. Implies '-x'. The preferred name for this option is '-L' as this is specified by POSIX.

--max-args=*max-args*
-n *max-args*
Use at most *max-args* arguments per command line. Fewer than *max-args* arguments will be used if the size (see the '-s' option) is exceeded, unless the '-x' option is given, in which case xargs will exit.

--dos-format
-D Convert the filenames to DOS format by replacing forward slashes with backslashes before submitting the filenames to the command. Implies a limit on the command-line length that is 127 characters plus the length of the command name.

`--max-chars=max-chars`
`-s max-chars`
: Use at most *max-chars* characters per command line, including the command initial arguments and the terminating nulls at the ends of the argument strings. If you specify a value for this option which is too large or small, a warning message is printed and the appropriate upper or lower limit is used instead. You can use '`--show-limits`' option to understand the command-line limits applying to `xargs` and how this is affected by any other options.

`--max-procs=max-procs`
`-P max-procs`
: Run up to *max-procs* processes at a time; the default is 1. If *max-procs* is 0, `xargs` will run as many processes as possible at a time. Use the '`-n`', '`-s`', or '`-L`' option with '`-P`'; otherwise chances are that the command will be run only once.

  On MS-DOS, you cannot run more than 1 process at a time.

When you run `xargs` on MS-DOS, the maxumum length of the arguments is further limited by the inherent restriction in the operating system call that invokes child programs. When the child program starts, it gets the command-line tail which cannot be longer than 126 characters. When DJGPP programs are invoked by `xargs`, they don't suffer from this limitation, so `xargs` does not automatically limit the maximum command-line length so that it won't get into the way of benefits from using DJGPP programs. It is your responsibility to know which programs can and which cannot get long command lines, and use the `-s 126` option with those which cannot. If any of the commands are invoked with `-dosexec` action (see Section 3.3.1 [Single File], page 23), `xargs` automatically impose the 126-character restriction. (Actually, only the tail of the command, without the command name itself, is limited to 126 characters, so `xargs` limits the command line to 126 + 1 blank + the length of the command name.)

3.3.2.5 Interspersing File Names

`xargs` can insert the name of the file it is processing between arguments you give for the command. Unless you also give options to limit the command size (see Section 3.3.2.4 [Limiting Command Size], page 28), this mode of operation is equivalent to '`find -exec`' (see Section 3.3.1 [Single File], page 23).

`--replace[=replace-str]`
`-I replace-str`
`-i replace-str`
: Replace occurrences of *replace-str* in the initial arguments with names read from the input. Also, unquoted blanks do not terminate arguments; instead, the input is split at newlines only. For the '`-i`' option, if *replace-str* is omitted for '`--replace`' or '`-i`', it defaults to '`{}`' (like for '`find -exec`'). Implies '`-x`' and '`-l 1`'. '`-i`' is deprecated in favour of '`-I`'. As an example, to sort each file in the `bills` directory, leaving the output in that file name with `.sorted` appended, you could do:

  ```
  find bills -type f | xargs -I XX sort -o XX.sorted XX
  ```

  The equivalent command using '`find -execdir`' is:

```
find bills -type f -execdir sort -o '{}.sorted' '{}' ';'
```

When you use the '-I' option, each line read from the input is buffered internally. This means that there is an upper limit on the length of input line that xargs will accept when used with the '-I' option. To work around this limitation, you can use the '-s' option to increase the amount of buffer space that xargs uses, and you can also use an extra invocation of xargs to ensure that very long lines do not occur. For example:

```
somecommand | xargs -s 50000 echo | xargs -I '{}' -s 100000 rm '{}'
```

Here, the first invocation of `xargs` has no input line length limit because it doesn't use the '-I' option. The second invocation of `xargs` does have such a limit, but we have ensured that the it never encounters a line which is longer than it can handle.

This is not an ideal solution. Instead, the '-I' option should not impose a line length limit (apart from any limit imposed by the operating system) and so one might consider this limitation to be a bug. A better solution would be to allow `xargs -I` to automatically move to a larger value for the '-s' option when this is needed.

This sort of problem doesn't occur with the output of `find` because it emits just one filename per line.

3.3.3 Querying

To ask the user whether to execute a command on a single file, you can use the `find` primary '-okdir' instead of '-execdir', and the `find` primary '-ok' instead of '-exec':

`-okdir` *command ;* [Action]

Like '-execdir' (see Section 3.3.1 [Single File], page 23), but ask the user first (on the standard input); if the response does not start with 'y' or 'Y', do not run the command, and return false. If the command is run, its standard input is redirected from `/dev/null`.

`-ok` *command ;* [Action]

This insecure variant of the '-okdir' action is specified by POSIX. The main difference is that the command is executed in the directory from which `find` was invoked, meaning that '{}' is expanded to a relative path starting with the name of one of the starting directories, rather than just the basename of the matched file. If the command is run, its standard input is redirected from `/dev/null`.

`-dosok` *command ;* [Action]

Like '-dosexec' (see Section 3.3.1 [Single File], page 23), but ask the user first (on the standard input); if the response does not start with 'y' or 'Y', do not run the command, and return false. Like '-dosexec', this action also affects all the other '-ok' and '-exec' (see Section 3.3.1 [Single File], page 23) actions for this invocation of `find`; so mixing '-exec', '-ok' and '-dosok' might be unwise unless the programs involved understand both kinds of slashes.

When processing multiple files with a single command, to query the user you give `xargs` the following option. When using this option, you might find it useful to control the number of files processed per invocation of the command (see Section 3.3.2.4 [Limiting Command Size], page 28).

`--interactive`
`-p` Prompt the user about whether to run each command line and read a line from the terminal. Only run the command line if the response starts with '`y`' or '`Y`'. Implies '`-t`'.

3.4 Delete Files

`-delete` [Action]

Delete files or directories; true if removal succeeded. If the removal failed, an error message is issued.

The use of the '`-delete`' action on the command line automatically turns on the '`-depth`' option (see Section 1.3 [find Expressions], page 3).

3.5 Adding Tests

You can test for file attributes that none of the `find` builtin tests check. To do this, use `xargs` to run a program that filters a list of files printed by `find`. If possible, use `find` builtin tests to pare down the list, so the program run by `xargs` has less work to do. The tests builtin to `find` will likely run faster than tests that other programs perform.

For reasons of efficiency it is often useful to limit the number of times an external program has to be run. For this reason, it is often a good idea to implement "extended" tests by using `xargs`.

For example, here is a way to print the names of all of the unstripped binaries in the `/usr/local` directory tree. Builtin tests avoid running `file` on files that are not regular files or are not executable.

```
find /usr/local -type f -perm /a=x | xargs file |
  grep 'not stripped' | cut -d: -f1
```

The `cut` program removes everything after the file name from the output of `file`.

However, using `xargs` can present important security problems (see Chapter 9 [Security Considerations], page 70). These can be avoided by using '`-execdir`'. The '`-execdir`' action is also a useful way of putting your own test in the middle of a set of other tests or actions for `find` (for example, you might want to use '`-prune`').

To place a special test somewhere in the middle of a `find` expression, you can use '`-execdir`' (or, less securely, '`-exec`') to run a program that performs the test. Because '`-execdir`' evaluates to the exit status of the executed program, you can use a program (which can be a shell script) that tests for a special attribute and make it exit with a true (zero) or false (non-zero) status. It is a good idea to place such a special test *after* the builtin tests, because it starts a new process which could be avoided if a builtin test evaluates to false.

Here is a shell script called `unstripped` that checks whether its argument is an unstripped binary file:

```
#! /bin/sh
file "$1" | grep -q "not stripped"
```

This script relies on the shell exiting with the status of the last command in the pipeline, in this case `grep`. The `grep` command exits with a true status if it found any matches, false if not. Here is an example of using the script (assuming it is in your search path). It lists the stripped executables (and shell scripts) in the file `sbins` and the unstripped ones in `ubins`.

```
find /usr/local -type f -perm /a=x \
  \( -execdir unstripped '{}' \; -fprint ubins -o -fprint sbins \)
```

Note that due to limitations of native MS-DOS shells, this technique is unavailable on MS-DOS (`COMMAND.COM` doesn't return the exit code of the last program it executed).

4 File Name Databases

The file name databases used by `locate` contain lists of files that were in particular directory trees when the databases were last updated. The file name of the default database is determined when `locate` and `updatedb` are configured and installed. The frequency with which the databases are updated and the directories for which they contain entries depend on how often `updatedb` is run, and with which arguments.

You can obtain some statistics about the databases by using '`locate --statistics`'.

4.1 Database Locations

There can be multiple file name databases. Users can select which databases `locate` searches using the `LOCATE_PATH` environment variable or a command line option. The system administrator can choose the file name of the default database, the frequency with which the databases are updated, and the directories for which they contain entries. File name databases are updated by running the `updatedb` program, typically nightly.

In networked environments, it often makes sense to build a database at the root of each filesystem, containing the entries for that filesystem. `updatedb` is then run for each filesystem on the fileserver where that filesystem is on a local disk, to prevent thrashing the network.

See Section 6.3 [Invoking updatedb], page 45, for the description of the options to `updatedb`, which specify which directories would each database contain entries for.

4.2 Database Formats

The file name databases contain lists of files that were in particular directory trees when the databases were last updated. The file name database format changed starting with GNU `locate` version 4.0 to allow machines with different byte orderings to share the databases. The new GNU `locate` can read both the old and new database formats. However, old versions of `locate` and `find` produce incorrect results if given a new-format database.

Support for the old database format will eventually be discontinued, first in `updatedb` and later in `xargs`.

If you run '`locate --statistics`', the resulting summary indicates the type of each `locate` database.

4.2.1 New Database Format

`updatedb` runs a program called `frcode` to *front-compress* the list of file names, which reduces the database size by a factor of 4 to 5. Front-compression (also known as incremental encoding) works as follows.

The database entries are a sorted list (case-insensitively, for users' convenience). Since the list is sorted, each entry is likely to share a prefix (initial string) with the previous entry. Each database entry begins with an offset-differential count byte, which is the additional number of characters of prefix of the preceding entry to use beyond the number that the preceding entry is using of its predecessor. (The counts can be negative.) Following the count is a null-terminated ASCII remainder—the part of the name that follows the shared prefix.

If the offset-differential count is larger than can be stored in a byte (+/-127), the byte has the value 0x80 and the count follows in a 2-byte word, with the high byte first (network byte order).

Every database begins with a dummy entry for a file called `LOCATE02`, which `locate` checks for to ensure that the database file has the correct format; it ignores the entry in doing the search.

Databases cannot be concatenated together, even if the first (dummy) entry is trimmed from all but the first database. This is because the offset-differential count in the first entry of the second and following databases will be wrong.

In the output of '`locate --statistics`', the new database format is referred to as '`LOCATE02`'.

4.2.2 Sample Database

Sample input to `frcode`:

```
/usr/src
/usr/src/cmd/aardvark.c
/usr/src/cmd/armadillo.c
/usr/tmp/zoo
```

Length of the longest prefix of the preceding entry to share:

```
0 /usr/src
8 /cmd/aardvark.c
14 rmadillo.c
5 tmp/zoo
```

Output from `frcode`, with trailing nulls changed to newlines and count bytes made printable:

```
0 LOCATE02
0 /usr/src
8 /cmd/aardvark.c
6 rmadillo.c
-9 tmp/zoo
```

(6 = 14 - 8, and -9 = 5 - 14)

4.2.3 Old Database Format

The old database format is used by Unix `locate` and `find` programs and earlier releases of the GNU ones. `updatedb` produces this format if given the '`--old-format`' option (supported by the MS-DOS `updatedb.bat` batch file).

Old versions of GNU `locate` fail to correctly handle very long file names, possibly leading to security problems relating to a heap buffer overrun. See Section 9.4 [Security Considerations for locate], page 75, for a detailed explanation.

`updatedb` runs programs called `bigram` and `code` to produce old-format databases. The old format differs from the new one in the following ways. Instead of each entry starting with an offset-differential count byte and ending with a null, byte values from 0 through 28 indicate offset-differential counts from -14 through 14. The byte value indicating that a long offset-differential count follows is 0x1e (30), not 0x80. The long counts are stored in host byte order, which is not necessarily network byte order, and host integer word size, which is usually 4 bytes. They also represent a count 14 less than their value. The database lines have no termination byte; the start of the next line is indicated by its first byte having a value <= 30.

In addition, instead of starting with a dummy entry, the old database format starts with a 256 byte table containing the 128 most common bigrams in the file list. A bigram is a pair of adjacent bytes. Bytes in the database that have the high bit set are indexes (with the high bit cleared) into the bigram table. The bigram and offset-differential count coding makes these databases 20-25% smaller than the new format, but makes them not 8-bit clean. Any byte in a file name that is in the ranges used for the special codes is replaced in the database by a question mark, which not coincidentally is the shell wildcard to match a single character.

The old format therefore cannot faithfully store entries with non-ASCII characters. It therefore should not be used in internationalised environments.

The output of '`locate --statistics`' will give an incorrect count of the number of file names containing newlines or high-bit characters for old-format databases.

4.3 Newline Handling

Within the database, file names are terminated with a null character. This is the case for both the old and the new format.

When the new database format is being used, the compression technique used to generate the database though relies on the ability to sort the list of files before they are presented to `frcode`.

If the system's sort command allows its input list of files to be separated with null characters via the '`-z`' option, this option is used and therefore `updatedb` and `locate` will both correctly handle file names containing newlines. If the `sort` command lacks support for this, the list of files is delimited with the newline character, meaning that parts of file names containing newlines will be incorrectly sorted. This can result in both incorrect matches and incorrect failures to match.

On the other hand, if you are using the old database format, file names with embedded newlines are not correctly handled. There is no technical limitation which enforces this, it's just that the `bigram` program has not been updated to support lists of file names separated by nulls.

So, if you are using the new database format (this is the default) and your system uses GNU `sort`, newlines will be correctly handled at all times. Otherwise, newlines may not be correctly handled.

5 File Permissions

Each file has a set of *permissions* that control the kinds of access that users have to that file. The permissions for a file are also called its *access mode*. They can be represented either in symbolic form or as an octal number.

5.1 Structure of File Permissions

There are three kinds of permissions that a user can have for a file:

1. permission to read the file. For directories, this means permission to list the contents of the directory.
2. permission to write to (change) the file. For directories, this means permission to create and remove files in the directory.
3. permission to execute the file (run it as a program). For directories, this means permission to access files in the directory.

There are three categories of users who may have different permissions to perform any of the above operations on a file:

1. the file's owner;
2. other users who are in the file's group;
3. everyone else.

Files are given an owner and group when they are created. Usually the owner is the current user and the group is the group of the directory the file is in, but this varies with the operating system, the file system the file is created on, and the way the file is created. You can change the owner and group of a file by using the `chown` and `chgrp` commands.

In addition to the three sets of three permissions listed above, a file's permissions have three special components, which affect only executable files (programs) and, on some systems, directories:

1. Set the process's effective user ID to that of the file upon execution (called the *setuid bit*). No effect on directories.
2. Set the process's effective group ID to that of the file upon execution (called the *setgid bit*). For directories on some systems, put files created in the directory into the same group as the directory, no matter what group the user who creates them is in.
3. prevent users from removing or renaming a file in a directory unless they own the file or the directory; this is called the *restricted deletion flag* for the directory. For regular files on some systems, save the program's text image on the swap device so it will load more quickly when run; this is called the *sticky bit*.

In addition to the permissions listed above, there may be file attributes specific to the file system, e.g: access control lists (ACLs), whether a file is compressed, whether a file can be modified (immutability), whether a file can be dumped. These are usually set using programs specific to the file system. For example:

ext2
: On GNU and GNU/Linux the file permissions ("attributes") specific to the ext2 file system are set using `chattr`.

FFS On FreeBSD the file permissions ("flags") specific to the FFS file system are set using `chrflags`.

Although a file's permission "bits" allow an operation on that file, that operation may still fail, because:

- the file-system-specific permissions do not permit it;
- the file system is mounted as read-only.

For example, if the immutable attribute is set on a file, it cannot be modified, regardless of the fact that you may have just run `chmod a+w FILE`.

5.2 Symbolic Modes

Symbolic modes represent changes to files' permissions as operations on single-character symbols. They allow you to modify either all or selected parts of files' permissions, optionally based on their previous values, and perhaps on the current `umask` as well (see Section 5.2.6 [Umask and Protection], page 40).

The format of symbolic modes is:

[ugoa...][+-=]*perms*...[,...]

where *perms* is either zero or more letters from the set '`rwxXst`', or a single letter from the set '`ugo`'.

The following sections describe the operators and other details of symbolic modes.

5.2.1 Setting Permissions

The basic symbolic operations on a file's permissions are adding, removing, and setting the permission that certain users have to read, write, and execute the file. These operations have the following format:

users operation permissions

The spaces between the three parts above are shown for readability only; symbolic modes cannot contain spaces.

The *users* part tells which users' access to the file is changed. It consists of one or more of the following letters (or it can be empty; see Section 5.2.6 [Umask and Protection], page 40, for a description of what happens then). When more than one of these letters is given, the order that they are in does not matter.

`u` the user who owns the file;

`g` other users who are in the file's group;

`o` all other users;

`a` all users; the same as '`ugo`'.

The *operation* part tells how to change the affected users' access to the file, and is one of the following symbols:

`+` to add the *permissions* to whatever permissions the *users* already have for the file;

`-` to remove the *permissions* from whatever permissions the *users* already have for the file;

`=` to make the *permissions* the only permissions that the *users* have for the file.

The *permissions* part tells what kind of access to the file should be changed; it is normally zero or more of the following letters. As with the *users* part, the order does not matter when more than one letter is given. Omitting the *permissions* part is useful only with the '`=`' operation, where it gives the specified *users* no access at all to the file.

`r` the permission the *users* have to read the file;

`w` the permission the *users* have to write to the file;

`x` the permission the *users* have to execute the file.

For example, to give everyone permission to read and write a file, but not to execute it, use:

```
a=rw
```

To remove write permission for all users other than the file's owner, use:

```
go-w
```

The above command does not affect the access that the owner of the file has to it, nor does it affect whether other users can read or execute the file.

To give everyone except a file's owner no permission to do anything with that file, use the mode below. Other users could still remove the file, if they have write permission on the directory it is in.

```
go=
```

Another way to specify the same thing is:

```
og-rwx
```

5.2.2 Copying Existing Permissions

You can base a file's permissions on its existing permissions. To do this, instead of using a series of '`r`', '`w`', or '`x`' letters after the operator, you use the letter '`u`', '`g`', or '`o`'. For example, the mode

```
o+g
```

adds the permissions for users who are in a file's group to the permissions that other users have for the file. Thus, if the file started out as mode 664 ('`rw-rw-r--`'), the above mode would change it to mode 666 ('`rw-rw-rw-`'). If the file had started out as mode 741 ('`rwxr----x`'), the above mode would change it to mode 745 ('`rwxr--r-x`'). The '`-`' and '`=`' operations work analogously.

5.2.3 Changing Special Permissions

In addition to changing a file's read, write, and execute permissions, you can change its special permissions. See Section 5.1 [Mode Structure], page 36, for a summary of these permissions.

To change a file's permission to set the user ID on execution, use '`u`' in the *users* part of the symbolic mode and '`s`' in the *permissions* part.

To change a file's permission to set the group ID on execution, use '`g`' in the *users* part of the symbolic mode and '`s`' in the *permissions* part.

To change a file's permission to set the restricted deletion flag or sticky bit, omit the *users* part of the symbolic mode (or use '`a`') and put '`t`' in the *permissions* part.

For example, to add set-user-ID permission to a program, you can use the mode:

```
u+s
```

To remove both set-user-ID and set-group-ID permission from it, you can use the mode:

```
ug-s
```

To set the restricted deletion flag or sticky bit, you can use the mode:

```
+t
```

The combination '`o+s`' has no effect. On GNU systems the combinations '`u+t`' and '`g+t`' have no effect, and '`o+t`' acts like plain '`+t`'.

The '`=`' operator is not very useful with special permissions; for example, the mode:

```
o=t
```

does set the restricted deletion flag or sticky bit, but it also removes all read, write, and execute permissions that users not in the file's group might have had for it.

5.2.4 Conditional Executability

There is one more special type of symbolic permission: if you use '`X`' instead of '`x`', execute permission is affected only if the file is a directory or already had execute permission.

For example, this mode:

```
a+X
```

gives all users permission to search directories, or to execute files if anyone could execute them before.

5.2.5 Making Multiple Changes

The format of symbolic modes is actually more complex than described above (see Section 5.2.1 [Setting Permissions], page 37). It provides two ways to make multiple changes to files' permissions.

The first way is to specify multiple *operation* and *permissions* parts after a *users* part in the symbolic mode.

For example, the mode:

```
og+rX-w
```

gives users other than the owner of the file read permission and, if it is a directory or if someone already had execute permission to it, gives them execute permission; and it also denies them write permission to the file. It does not affect the permission that the owner of the file has for it. The above mode is equivalent to the two modes:

```
og+rX
og-w
```

The second way to make multiple changes is to specify more than one simple symbolic mode, separated by commas. For example, the mode:

```
a+r,go-w
```

gives everyone permission to read the file and removes write permission on it for all users except its owner. Another example:

```
u=rwx,g=rx,o=
```

sets all of the non-special permissions for the file explicitly. (It gives users who are not in the file's group no permission at all for it.)

The two methods can be combined. The mode:

```
a+r,g+x-w
```

gives all users permission to read the file, and gives users who are in the file's group permission to execute it, as well, but not permission to write to it. The above mode could be written in several different ways; another is:

```
u+r,g+rx,o+r,g-w
```

5.2.6 The Umask and Protection

If the *users* part of a symbolic mode is omitted, it defaults to '`a`' (affect all users), except that any permissions that are *set* in the system variable `umask` are *not affected*. The value of `umask` can be set using the `umask` command. Its default value varies from system to system.

Omitting the *users* part of a symbolic mode is generally not useful with operations other than '`+`'. It is useful with '`+`' because it allows you to use `umask` as an easily customizable protection against giving away more permission to files than you intended to.

As an example, if `umask` has the value 2, which removes write permission for users who are not in the file's group, then the mode:

```
+w
```

adds permission to write to the file to its owner and to other users who are in the file's group, but *not* to other users. In contrast, the mode:

```
a+w
```

ignores `umask`, and *does* give write permission for the file to all users.

5.3 Numeric Modes

As an alternative to giving a symbolic mode, you can give an octal (base 8) number that represents the new mode. This number is always interpreted in octal; you do not have to add a leading 0, as you do in C. Mode 0055 is the same as mode 55.

A numeric mode is usually shorter than the corresponding symbolic mode, but it is limited in that it cannot take into account a file's previous permissions; it can only set them absolutely.

The permissions granted to the user, to other users in the file's group, and to other users not in the file's group each require three bits, which are represented as one octal digit. The three special permissions also require one bit each, and they are as a group represented as another octal digit. Here is how the bits are arranged, starting with the lowest valued bit:

```
Value in  Corresponding
Mode      Permission

          Other users not in the file's group:
   1      Execute
   2      Write
```

```
   4      Read

          Other users in the file's group:
  10      Execute
  20      Write
  40      Read

          The file's owner:
 100      Execute
 200      Write
 400      Read

          Special permissions:
1000      Restricted deletion flag or sticky bit
2000      Set group ID on execution
4000      Set user ID on execution
```

For example, numeric mode 4755 corresponds to symbolic mode 'u=rwxs,go=rx', and numeric mode 664 corresponds to symbolic mode 'ug=rw,o=r'. Numeric mode 0 corresponds to symbolic mode 'a='.

6 Reference

Below are summaries of the command line syntax for the programs discussed in this manual.

6.1 Invoking `find`

`find` [-H] [-L] [-P] [*file...*] [*expression*]

`find` searches the directory tree rooted at each file name *file* by evaluating the *expression* on each file it finds in the tree.

The options '`-H`', '`-L`' or '`-P`' may be specified at the start of the command line (if none of these is specified, '`-P`' is assumed). If you specify more than one of these options, the last one specified takes effect (but note that the '`-follow`' option is equivalent to '`-L`'). The arguments after these are a list of files or directories that should be searched.

This list of files to search is followed by a list of expressions describing the files we wish to search for. The first part of the expression is recognised by the fact that it begins with '`-`', '`(`', '`)`', '`,`', or '`!`'. Any arguments after it are the rest of the expression. If no files are given, the current directory is used. If no expression is given, the expression '`-print`' is used.

`find` exits with status zero if all files matched are processed successfully, greater than 0 if errors occur.

Three options can precede the list of files. They determine the way that symbolic links are handled.

`-P` Never follow symbolic links (this is the default), except in the case of the '`-xtype`' predicate.

`-L` Always follow symbolic links, except in the case of the '`-xtype`' predicate.

`-H` Follow symbolic links specified in the list of files to search, or which are otherwise specified on the command line.

If `find` would follow a symbolic link, but cannot for any reason (for example, because it has insufficient permissions or the link is broken), it falls back on using the properties of the symbolic link itself. Section 2.2.1 [Symbolic Links], page 7 for a more complete description of how symbolic links are handled.

See [Primary Index], page 87, for a summary of all of the tests, actions, and options that the expression can contain. If the expression is missing, '`-print`' is assumed.

`find` also recognises two options for administrative use:

`--help` Print a summary of the command line usage and exit.

`--version`
`-version` Print the version number of `find` and exit.

6.1.1 Warning Messages

If there is an error on the `find` command line, an error message is normally issued. However, there are some usages that are inadvisable but which `find` should still accept. Under these circumstances, `find` may issue a warning message. By default, warnings are enabled only if `find` is being run interactively (specifically, if the standard input is a terminal). Warning messages can be controlled explicitly by the use of options on the command line:

`-warn` Issue warning messages where appropriate.

`-nowarn` Do not issue warning messages.

These options take effect at the point on the command line where they are specified. Therefore it's not useful to specify '`-nowarn`' at the end of the command line. The warning messages affected by the above options are triggered by:

- Use of the '`-d`' option which is deprecated; please use '`-depth`' instead, since the latter is POSIX-compliant.
- Use of the '`-ipath`' option which is deprecated; please use '`-iwholename`' instead.
- Specifying an option (for example '`-mindepth`') after a non-option (for example '`-type`' or '`-print`') on the command line.
- Use of the '`-name`' or '`-iname`' option with a slash character in the pattern. Since the name predicates only compare against the basename of the visited files, the only file that can match a slash is the root directory itself.

The default behaviour above is designed to work in that way so that existing shell scripts don't generate spurious errors, but people will be made aware of the problem.

Some warning messages are issued for less common or more serious problems, and consequently cannot be turned off:

- Use of an unrecognised backslash escape sequence with '`-fprintf`'
- Use of an unrecognised formatting directive with '`-fprintf`'

6.2 Invoking `locate`

`locate` [*option*...] *pattern*...

For each *pattern* given `locate` searches one or more file name databases returning each match of *pattern*.

For each *pattern* given `locate` searches one or more file name databases returning each match of *pattern*.

`--all`
`-A` Print only names which match all non-option arguments, not those matching one or more non-option arguments.

`--basename`
`-b` The specified pattern is matched against just the last component of the name of a file in the `locate` database. This last component is also called the "base name". For example, the base name of `/tmp/mystuff/foo.old.c` is `foo.old.c`. If the pattern contains metacharacters, it must match the base name exactly. If not, it must match part of the base name.

`--count`
`-c` Instead of printing the matched file names, just print the total number of matches found, unless '`--print`' ('`-p`') is also present.

`--database=`*path*
`-d` *path* Instead of searching the default `locate` database, `locate` search the file name databases in *path*, which is a colon-separated list of database file names. You

can also use the environment variable `LOCATE_PATH` to set the list of database files to search. The option overrides the environment variable if both are used. Empty elements in *path* (that is, a leading or trailing colon, or two colons in a row) are taken to stand for the default database. A database can be supplied on stdin, using '`-`' as an element of '`path`'. If more than one element of '`path`' is '`-`', later instances are ignored (but a warning message is printed).

`--existing`
`-e` Only print out such names which currently exist (instead of such names which existed when the database was created). Note that this may slow down the program a lot, if there are many matches in the database. The way in which broken symbolic links are treated is affected by the '`-L`', '`-P`' and '`-H`' options. Please note that it is possible for the file to be deleted after `locate` has checked that it exists, but before you use it.

`--non-existing`
`-E` Only print out such names which currently do not exist (instead of such names which existed when the database was created). Note that this may slow down the program a lot, if there are many matches in the database. The way in which broken symbolic links are treated is affected by the '`-L`', '`-P`' and '`-H`' options. Please note that `locate` checks that the file does not exist, but a file of the same name might be created after `locate`'s check but before you read `locate`'s output.

`--follow`
`-L` If testing for the existence of files (with the '`-e`' or '`-E`' options), consider broken symbolic links to be non-existing. This is the default behaviour.

`--nofollow`
`-P`
`-H` If testing for the existence of files (with the '`-e`' or '`-E`' options), treat broken symbolic links as if they were existing files. The '`-H`' form of this option is provided purely for similarity with `find`; the use of '`-P`' is recommended over '`-H`'.

`--ignore-case`
`-i` Ignore case distinctions in both the pattern and the file names.

`--limit=N`
`-l N` Limit the number of results printed to N. When used with the '`--count`' option, the value printed will never be larger than this limit.

`--mmap`
`-m` Accepted but does nothing. The option is supported only to provide compatibility with BSD's `locate`.

`--null`
`-0` Results are separated with the ASCII NUL character rather than the newline character. To get the full benefit of the use of this option, use the new `locate` database format (that is the default anyway).

`--print`
`-p`
: Print search results when they normally would not, because of the presence of '`--statistics`' ('`-S`') or '`--count`' ('`-c`').

`--wholename`
`-w`
: The specified pattern is matched against the whole name of the file in the `locate` database. If the pattern contains metacharacters, it must match exactly. If not, it must match part of the whole file name. This is the default behaviour.

`--regex`
`-r`
: Instead of using substring or shell glob matching, the pattern specified on the command line is understood to be a regular expression. GNU Emacs-style regular expressions are assumed unless the '`--regextype`' option is also given. File names from the `locate` database are matched using the specified regular expression. If the '`-i`' flag is also given, matching is case-insensitive. Matches are performed against the whole path name, and so by default a pathname will be matched if any part of it matches the specified regular expression. The regular expression may use '`^`' or '`$`' to anchor a match at the beginning or end of a pathname.

`--regextype`
: This option changes the regular expression syntax and behaviour used by the '`--regex`' option. Section 6.5 [Regular Expressions], page 49 for more information on the regular expression dialects understood by GNU findutils.

`--stdio`
`-s`
: Accepted but does nothing. The option is supported only to provide compatibility with BSD's `locate`.

`--statistics`
`-S`
: Print some summary information for each `locate` database. No search is performed unless non-option arguments are given. Although the BSD version of locate also has this option, the format of the output is different.

`--help`
: Print a summary of the command line usage for `locate` and exit.

`--version`
: Print the version number of `locate` and exit.

6.3 Invoking `updatedb`

`updatedb [`*option*...`]`

`updatedb` creates and updates the database of file names used by `locate`. `updatedb` generates a list of files similar to the output of `find` and then uses utilities for optimizing the database for performance. `updatedb` is often run periodically as a `cron` job and configured with environment variables or command options. Typically, operating systems have a shell script that "exports" configurations for variable definitions and uses another shell script that "sources" the configuration file into the environment and then executes `updatedb` in the environment.

`updatedb` creates and updates the database of file names used by `locate`. `updatedb` generates a list of files similar to the output of `find` and then uses utilities for optimizing the

database for performance. `updatedb` is often run periodically as a `cron` job and configured with environment variables or command options. Typically, operating systems have a shell script that "exports" configurations for variable definitions and uses another shell script that "sources" the configuration file into the environment and then executes `updatedb` in the environment.

`--findoptions='`*`OPTION...`*`'`
: Global options to pass on to `find`. The environment variable `FINDOPTIONS` also sets this value. Default is none.

`--localpaths='`*`path...`*`'`
: Non-network directories to put in the database. Default is `/`.

`--netpaths='`*`path...`*`'`
: Network (NFS, AFS, RFS, etc.) directories to put in the database. The environment variable `NETPATHS` also sets this value. Default is none.

`--prunepaths='`*`path...`*`'`
: Directories to omit from the database, which would otherwise be included. The environment variable `PRUNEPATHS` also sets this value. Default is `/tmp /usr/tmp /var/tmp /afs`. The paths are used as regular expressions (with `find ... -regex`, so you need to specify these paths in the same way that `find` will encounter them. This means for example that the paths must not include trailing slashes.

`--prunefs='`*`path...`*`'`
: Filesystems to omit from the database, which would otherwise be included. Note that files are pruned when a filesystem is reached; Any filesystem mounted under an undesired filesystem will be ignored. The environment variable `PRUNEFS` also sets this value. Default is `nfs NFS proc`.

`--output=`*`dbfile`*
: The database file to build. Default is system-dependent, but typically `/usr/local/var/locatedb`.

`--localuser=`*`user`*
: The user to search the non-network directories as, using `su`. Default is to search the non-network directories as the current user. You can also use the environment variable `LOCALUSER` to set this user.

`--netuser=`*`user`*
: The user to search network directories as, using `su`. Default `user` is `daemon`. You can also use the environment variable `NETUSER` to set this user.

  Note that the simplified `updatedb.bat` batch file used on MS-DOS doesn't support these options.

`--old-format`
: Generate a `locate` database in the old format, for compatibility with versions of `locate` other than GNU `locate`. Using this option means that `locate` will not be able to properly handle non-ASCII characters in file names (that is, file names containing characters which have the eighth bit set, such as many of the characters from the ISO-8859-1 character set). Versions of `locate` prior to

4.2.31 fail to correctly handle long filenames in old-format databases. Versions 4.3.0 to 4.3.6 were also affected.

`--help` Print a summary of the command line usage and exit.

`--version`
: Print the version number of `updatedb` and exit.

6.4 Invoking xargs

`xargs [option...] [command [initial-arguments]]`

`xargs` exits with the following status:

0 if it succeeds

123 if any invocation of the command exited with status 1-125

124 if the command exited with status 255

125 if the command is killed by a signal

126 if the command cannot be run

127 if the command is not found

1 if some other error occurred.

Exit codes greater than 128 are used by the shell to indicate that a program died due to a fatal signal.

`--arg-file=`*inputfile*
`-a o`*inputfile*
: Read names from the file *inputfile* instead of standard input. If you use this option, the standard input stream remains unchanged when commands are run. Otherwise, stdin is redirected from `/dev/null`.

`--null`
`-0` Input file names are terminated by a null character instead of by whitespace, and any quotes and backslash characters are not considered special (every character is taken literally). Disables the end of file string, which is treated like any other argument.

`--delimiter` *delim*
`-d` *delim*
: Input file names are terminated by the specified character *delim* instead of by whitespace, and any quotes and backslash characters are not considered special (every character is taken literally). Disables the end of file string, which is treated like any other argument.

The specified delimiter may be a single character, a C-style character escape such as '`\n`', or an octal or hexadecimal escape code. Octal and hexadecimal escape codes are understood as for the `printf` command. Multibyte characters are not supported.

`-E` *eof-str*
`--eof`[=*eof-str*]
`-e`[*eof-str*]
: Set the end of file string to *eof-str*. If the end of file string occurs as a line of input, the rest of the input is ignored. If *eof-str* is omitted ('`-e`') or blank (either '`-e`' or '`-E`'), there is no end of file string. The '`-e`' form of this option is deprecated in favour of the POSIX-compliant '`-E`' option, which you should use instead. As of GNU xargs version 4.2.9, the default behaviour of xargs is not to have a logical end-of-file marker. The POSIX standard (IEEE Std 1003.1, 2004 Edition) allows this.

`--help`
: Print a summary of the options to `xargs` and exit.

`-I` *replace-str*
`--replace`[=*replace-str*]
`-i`[*replace-str*]
: Replace occurrences of *replace-str* in the initial arguments with names read from standard input. Also, unquoted blanks do not terminate arguments; instead, the input is split at newlines only. If *replace-str* is omitted (omitting it is allowed only for '`-i`'), it defaults to '`{}`' (like for '`find -exec`'). Implies '`-x`' and '`-l 1`'. The '`-i`' option is deprecated in favour of the '`-I`' option.

`-L` *max-lines*
`--max-lines`[=*max-lines*]
`-l`[*max-lines*]
: Use at most *max-lines* non-blank input lines per command line. For '`-l`', *max-lines* defaults to 1 if omitted. For '`-L`', the argument is mandatory. Trailing blanks cause an input line to be logically continued on the next input line, for the purpose of counting the lines. Implies '`-x`'. The '`-l`' form of this option is deprecated in favour of the POSIX-compliant '`-L`' option.

`--max-args=`*max-args*
`-n` *max-args*
: Use at most *max-args* arguments per command line. Fewer than *max-args* arguments will be used if the size (see the '`-s`' option) is exceeded, unless the '`-x`' option is given, in which case `xargs` will exit.

`--interactive`
`-p`
: Prompt the user about whether to run each command line and read a line from the terminal. Only run the command line if the response starts with '`y`' or '`Y`'. Implies '`-t`'.

`--no-run-if-empty`
`-r`
: If the standard input is completely empty, do not run the command. By default, the command is run once even if there is no input.

`--max-chars=`*max-chars*
`-s` *max-chars*
: Use at most *max-chars* characters per command line, including the command, initial arguments and any terminating nulls at the ends of the argument strings.

`--show-limits`
: Display the limits on the command-line length which are imposed by the operating system, `xargs`' choice of buffer size and the '`-s`' option. Pipe the input from `/dev/null` (and perhaps specify '`--no-run-if-empty`') if you don't want `xargs` to do anything.

`--verbose`
`-t`
: Print the command line on the standard error output before executing it.

`--version`
: Print the version number of `xargs` and exit.

`--exit`
`-x`
: Exit if the size (see the '`-s`' option) is exceeded.

`--max-procs=`*max-procs*
`-P` *max-procs*
: Run simultaneously up to *max-procs* processes at once; the default is 1. If *max-procs* is 0, `xargs` will run as many processes as possible simultaneously.

6.5 Regular Expressions

The '`-regex`' and '`-iregex`' tests of `find` allow matching by regular expression, as does the '`--regex`' option of `locate`. There are many different types of Regular Expression, but the type used by `find` and `locate` is the same as is used in GNU Emacs. Both programs provide an option which allows you to select an alternative regular expression syntax; for `find` this is the '`-regextype`' option, and for `locate` this is the '`--regextype`' option.

These options take a single argument, which indicates the specific regular expression syntax and behaviour that should be used. This should be one of the following:

6.5.1 '`findutils-default`' regular expression syntax

The character '`.`' matches any single character.

'`+`'
: indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

'`?`'
: indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

'`\+`'
: matches a '`+`'

'`\?`'
: matches a '`?`'.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example '`[z-a]`', are ignored. Within square brackets, '`\`' is taken literally. Character classes are not supported, so for example you would need to use '`[0-9]`' instead of '`[[:digit:]]`'.

GNU extensions are supported:

1. '`\w`' matches a character within a word
2. '`\W`' matches a character which is not within a word
3. '`\<`' matches the beginning of a word

4. ‘\>’ matches the end of a word
5. ‘\b’ matches a word boundary
6. ‘\B’ matches characters which are not a word boundary
7. ‘\`’ matches the beginning of the whole input
8. ‘\'’ matches the end of the whole input

Grouping is performed with backslashes followed by parentheses ‘\(’, ‘\)’. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example ‘\2’ matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis ‘\(’.

The alternation operator is ‘\|’.

The character ‘^’ only represents the beginning of a string when it appears:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘\(’
3. After the alternation operator ‘\|’

The character ‘$’ only represents the end of a string when it appears:

1. At the end of a regular expression
2. Before a close-group, signified by ‘\)’
3. Before the alternation operator ‘\|’

‘*’, ‘+’ and ‘?’ are special at any point in a regular expression except:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘\(’
3. After the alternation operator ‘\|’

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.2 ‘awk’ regular expression syntax

The character ‘.’ matches any single character except the null character.

‘+’ indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

‘?’ indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

‘\+’ matches a ‘+’

‘\?’ matches a ‘?’.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example ‘[z-a]’, are invalid. Within square brackets, ‘\’ can be used to quote the following character. Character classes are not supported, so for example you would need to use ‘[0-9]’ instead of ‘[[:digit:]]’.

GNU extensions are not supported and so ‘\w’, ‘\W’, ‘\<’, ‘\>’, ‘\b’, ‘\B’, ‘\`’, and ‘\'’ match ‘w’, ‘W’, ‘<’, ‘>’, ‘b’, ‘B’, ‘`’, and ‘'’ respectively.

Grouping is performed with parentheses ‘()’. An unmatched ‘)’ matches just itself. A backslash followed by a digit matches that digit.

The alternation operator is ‘|’.

The characters ‘^’ and ‘$’ always represent the beginning and end of a string respectively, except within square brackets. Within brackets, ‘^’ can be used to invert the membership of the character class being specified.

‘*’, ‘+’ and ‘?’ are special at any point in a regular expression except:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘(’
3. After the alternation operator ‘|’

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.3 ‘egrep’ regular expression syntax

The character ‘.’ matches any single character except newline.

‘+’ indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

‘?’ indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

‘\+’ matches a ‘+’

‘\?’ matches a ‘?’.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example ‘[z-a]’, are ignored. Within square brackets, ‘\’ is taken literally. Character classes are supported; for example ‘[[:digit:]]’ will match a single decimal digit. Non-matching lists ‘[^...]’ do not ever match newline.

GNU extensions are supported:

1. ‘\w’ matches a character within a word
2. ‘\W’ matches a character which is not within a word
3. ‘\<’ matches the beginning of a word
4. ‘\>’ matches the end of a word
5. ‘\b’ matches a word boundary
6. ‘\B’ matches characters which are not a word boundary
7. ‘\‘’ matches the beginning of the whole input
8. ‘\’’ matches the end of the whole input

Grouping is performed with parentheses ‘()’. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example ‘\2’ matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis ‘(’.

The alternation operator is ‘|’.

The characters ‘^’ and ‘$’ always represent the beginning and end of a string respectively, except within square brackets. Within brackets, ‘^’ can be used to invert the membership of the character class being specified.

The characters ‘*’, ‘+’ and ‘?’ are special anywhere in a regular expression.

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.4 ‘emacs’ regular expression syntax

The character ‘.’ matches any single character except newline.

‘+’ indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

‘?’ indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

‘\+’ matches a ‘+’

‘\?’ matches a ‘?’.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example ‘[z-a]’, are ignored. Within square brackets, ‘\’ is taken literally. Character classes are not supported, so for example you would need to use ‘[0-9]’ instead of ‘[[:digit:]]’.

GNU extensions are supported:

1. ‘\w’ matches a character within a word
2. ‘\W’ matches a character which is not within a word
3. ‘\<’ matches the beginning of a word
4. ‘\>’ matches the end of a word
5. ‘\b’ matches a word boundary
6. ‘\B’ matches characters which are not a word boundary
7. ‘\‘’ matches the beginning of the whole input
8. ‘\’’ matches the end of the whole input

Grouping is performed with backslashes followed by parentheses ‘\(’, ‘\)’. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example ‘\2’ matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis ‘\(’.

The alternation operator is ‘\|’.

The character ‘^’ only represents the beginning of a string when it appears:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘\(’
3. After the alternation operator ‘\|’

The character ‘$’ only represents the end of a string when it appears:

1. At the end of a regular expression

2. Before a close-group, signified by '\)'
3. Before the alternation operator '\|'

'*', '+' and '?' are special at any point in a regular expression except:

1. At the beginning of a regular expression
2. After an open-group, signified by '\('
3. After the alternation operator '\|'

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.5 'gnu-awk' regular expression syntax

The character '.' matches any single character.

'+' indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

'?' indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

'\+' matches a '+'

'\?' matches a '?'.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example '[z-a]', are invalid. Within square brackets, '\' can be used to quote the following character. Character classes are supported; for example '[[:digit:]]' will match a single decimal digit.

GNU extensions are supported:

1. '\w' matches a character within a word
2. '\W' matches a character which is not within a word
3. '\<' matches the beginning of a word
4. '\>' matches the end of a word
5. '\b' matches a word boundary
6. '\B' matches characters which are not a word boundary
7. '\`' matches the beginning of the whole input
8. '\'' matches the end of the whole input

Grouping is performed with parentheses '()'. An unmatched ')' matches just itself. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example '\2' matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis '('.

The alternation operator is '|'.

The characters '^' and '$' always represent the beginning and end of a string respectively, except within square brackets. Within brackets, '^' can be used to invert the membership of the character class being specified.

'*', '+' and '?' are special at any point in a regular expression except:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘(’
3. After the alternation operator ‘|’

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.6 ‘grep’ regular expression syntax

The character ‘.’ matches any single character except newline.

‘\+’ indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

‘\?’ indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

‘+ and ?’ match themselves.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example ‘[z-a]’, are ignored. Within square brackets, ‘\’ is taken literally. Character classes are supported; for example ‘[[:digit:]]’ will match a single decimal digit. Non-matching lists ‘[^...]’ do not ever match newline.

GNU extensions are supported:

1. ‘\w’ matches a character within a word
2. ‘\W’ matches a character which is not within a word
3. ‘\<’ matches the beginning of a word
4. ‘\>’ matches the end of a word
5. ‘\b’ matches a word boundary
6. ‘\B’ matches characters which are not a word boundary
7. ‘\`’ matches the beginning of the whole input
8. ‘\'’ matches the end of the whole input

Grouping is performed with backslashes followed by parentheses ‘\(’, ‘\)’. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example ‘\2’ matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis ‘\(’.

The alternation operator is ‘\|’.

The character ‘^’ only represents the beginning of a string when it appears:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘\(’
3. After a newline
4. After the alternation operator ‘\|’

The character ‘$’ only represents the end of a string when it appears:

1. At the end of a regular expression

2. Before a close-group, signified by '\)'
3. Before a newline
4. Before the alternation operator '\|'

'*', '\+' and '\?' are special at any point in a regular expression except:

1. At the beginning of a regular expression
2. After an open-group, signified by '\('
3. After a newline
4. After the alternation operator '\|'

Intervals are specified by '\{' and '\}'. Invalid intervals such as 'a\{1z' are not accepted.

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.7 'posix-awk' regular expression syntax

The character '.' matches any single character except the null character.

'+' indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

'?' indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

'\+' matches a '+'

'\?' matches a '?'.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example '[z-a]', are invalid. Within square brackets, '\' can be used to quote the following character. Character classes are supported; for example '[[:digit:]]' will match a single decimal digit.

GNU extensions are not supported and so '\w', '\W', '\<', '\>', '\b', '\B', '\`', and '\'' match 'w', 'W', '<', '>', 'b', 'B', '`', and ''' respectively.

Grouping is performed with parentheses '()'. An unmatched ')' matches just itself. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example '\2' matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis '('.

The alternation operator is '|'.

The characters '^' and '$' always represent the beginning and end of a string respectively, except within square brackets. Within brackets, '^' can be used to invert the membership of the character class being specified.

'*', '+' and '?' are special at any point in a regular expression except the following places, where they are not allowed:

1. At the beginning of a regular expression
2. After an open-group, signified by '('
3. After the alternation operator '|'

Intervals are specified by ‘{’ and ‘}’. Invalid intervals such as ‘a{1z’ are not accepted.

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.8 ‘posix-basic’ regular expression syntax

The character ‘.’ matches any single character except the null character.

‘\+’ indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

‘\?’ indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

‘+ and ?’ match themselves.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example ‘[z-a]’, are invalid. Within square brackets, ‘\’ is taken literally. Character classes are supported; for example ‘[[:digit:]]’ will match a single decimal digit.

GNU extensions are supported:

1. ‘\w’ matches a character within a word
2. ‘\W’ matches a character which is not within a word
3. ‘\<’ matches the beginning of a word
4. ‘\>’ matches the end of a word
5. ‘\b’ matches a word boundary
6. ‘\B’ matches characters which are not a word boundary
7. ‘\`’ matches the beginning of the whole input
8. ‘\'’ matches the end of the whole input

Grouping is performed with backslashes followed by parentheses ‘\(’, ‘\)’. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example ‘\2’ matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis ‘\(’.

The alternation operator is ‘\|’.

The character ‘^’ only represents the beginning of a string when it appears:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘\(’
3. After the alternation operator ‘\|’

The character ‘$’ only represents the end of a string when it appears:

1. At the end of a regular expression
2. Before a close-group, signified by ‘\)’
3. Before the alternation operator ‘\|’

‘*’, ‘\+’ and ‘\?’ are special at any point in a regular expression except:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘\(’
3. After the alternation operator ‘\|’

Intervals are specified by ‘\{’ and ‘\}’. Invalid intervals such as ‘a\{1z’ are not accepted.

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.9 ‘posix-egrep’ regular expression syntax

The character ‘.’ matches any single character except newline.

‘+’ indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

‘?’ indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

‘\+’ matches a ‘+’

‘\?’ matches a ‘?’.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example ‘[z-a]’, are ignored. Within square brackets, ‘\’ is taken literally. Character classes are supported; for example ‘[[:digit:]]’ will match a single decimal digit. Non-matching lists ‘[^...]’ do not ever match newline.

GNU extensions are supported:

1. ‘\w’ matches a character within a word
2. ‘\W’ matches a character which is not within a word
3. ‘\<’ matches the beginning of a word
4. ‘\>’ matches the end of a word
5. ‘\b’ matches a word boundary
6. ‘\B’ matches characters which are not a word boundary
7. ‘\‘’ matches the beginning of the whole input
8. ‘\’’ matches the end of the whole input

Grouping is performed with parentheses ‘()’. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example ‘\2’ matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis ‘(’.

The alternation operator is ‘|’.

The characters ‘^’ and ‘$’ always represent the beginning and end of a string respectively, except within square brackets. Within brackets, ‘^’ can be used to invert the membership of the character class being specified.

The characters ‘*’, ‘+’ and ‘?’ are special anywhere in a regular expression.

Intervals are specified by ‘{’ and ‘}’. Invalid intervals are treated as literals, for example ‘a{1’ is treated as ‘a\{1’

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.5.10 ‘posix-extended’ regular expression syntax

The character ‘.’ matches any single character except the null character.

‘+’ indicates that the regular expression should match one or more occurrences of the previous atom or regexp.

‘?’ indicates that the regular expression should match zero or one occurrence of the previous atom or regexp.

‘\+’ matches a ‘+’

‘\?’ matches a ‘?’.

Bracket expressions are used to match ranges of characters. Bracket expressions where the range is backward, for example ‘[z-a]’, are invalid. Within square brackets, ‘\’ is taken literally. Character classes are supported; for example ‘[[:digit:]]’ will match a single decimal digit.

GNU extensions are supported:

1. ‘\w’ matches a character within a word
2. ‘\W’ matches a character which is not within a word
3. ‘\<’ matches the beginning of a word
4. ‘\>’ matches the end of a word
5. ‘\b’ matches a word boundary
6. ‘\B’ matches characters which are not a word boundary
7. ‘\‘’ matches the beginning of the whole input
8. ‘\’’ matches the end of the whole input

Grouping is performed with parentheses ‘()’. An unmatched ‘)’ matches just itself. A backslash followed by a digit acts as a back-reference and matches the same thing as the previous grouped expression indicated by that number. For example ‘\2’ matches the second group expression. The order of group expressions is determined by the position of their opening parenthesis ‘(’.

The alternation operator is ‘|’.

The characters ‘^’ and ‘$’ always represent the beginning and end of a string respectively, except within square brackets. Within brackets, ‘^’ can be used to invert the membership of the character class being specified.

‘*’, ‘+’ and ‘?’ are special at any point in a regular expression except the following places, where they are not allowed:

1. At the beginning of a regular expression
2. After an open-group, signified by ‘(’
3. After the alternation operator ‘|’

Intervals are specified by ‘{’ and ‘}’. Invalid intervals such as ‘a{1z’ are not accepted.

The longest possible match is returned; this applies to the regular expression as a whole and (subject to this constraint) to subexpressions within groups.

6.6 Environment Variables

LANG Provides a default value for the internationalisation variables that are unset or null.

LC_ALL If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
The POSIX standard specifies that this variable affects the pattern matching to be used for the '\-name' option. GNU find uses the GNU version of the `fnmatch` library function.

POSIX also specifies that the 'LC_COLLATE' environment variable affects the interpretation of the user's response to the query issued by '\-ok', but this is not the case for GNU find.

LC_CTYPE
This variable affects the treatment of character classes used with the '`-name`' test, if the system's `fnmatch` library function supports this. It has no effect on the behaviour of the '`-ok`' expression.

LC_MESSAGES
Determines the locale to be used for internationalised messages.

NLSPATH
Determines the location of the internationalisation message catalogues.

PATH Affects the directories which are searched to find the executables invoked by '`-exec`', '`-execdir`' '`-ok`' and '`-okdir`'. If the *PATH* environment variable includes the current directory (by explicitly including '`.`' or by having an empty element), and the find command line includes '`-execdir`' or '`-okdir`', `find` will refuse to run. See Chapter 9 [Security Considerations], page 70, for a more detailed discussion of security matters.

POSIXLY_CORRECT
Determines the block size used by '`-ls`' and '`-fls`'. If *POSIXLY_CORRECT* is set, blocks are units of 512 bytes. Otherwise they are units of 1024 bytes.

TZ Affects the time zone used for some of the time-related format directives of '`-printf`' and '`-fprintf`'.

7 Common Tasks

The sections that follow contain some extended examples that both give a good idea of the power of these programs, and show you how to solve common real-world problems.

7.1 Viewing And Editing

To view a list of files that meet certain criteria, simply run your file viewing program with the file names as arguments. Shells substitute a command enclosed in backquotes with its output, so the whole command looks like this:

```
less `find /usr/include -name '*.h' | xargs grep -l mode_t`
```

You can edit those files by giving an editor name instead of a file viewing program:

```
emacs `find /usr/include -name '*.h' | xargs grep -l mode_t`
```

Because there is a limit to the length of any individual command line, there is a limit to the number of files that can be handled in this way. We can get around this difficulty by using xargs like this:

```
find /usr/include -name '*.h' | xargs grep -l mode_t > todo
xargs --arg-file=todo emacs
```

Here, `xargs` will run `emacs` as many times as necessary to visit all of the files listed in the file `todo`.

7.2 Archiving

You can pass a list of files produced by `find` to a file archiving program. GNU `tar` and `cpio` can both read lists of file names from the standard input—either delimited by nulls (the safe way) or by blanks (the lazy, risky default way). To use null-delimited names, give them the '`--null`' option. You can store a file archive in a file, write it on a tape, or send it over a network to extract on another machine.

One common use of `find` to archive files is to send a list of the files in a directory tree to `cpio`. Use '`-depth`' so if a directory does not have write permission for its owner, its contents can still be restored from the archive since the directory's permissions are restored after its contents. Here is an example of doing this using `cpio`; you could use a more complex `find` expression to archive only certain files.

```
find . -depth -print0 |
  cpio --create --null --format=crc --file=/dev/nrst0
```

You could restore that archive using this command:

```
cpio --extract --null --make-dir --unconditional \
  --preserve --file=/dev/nrst0
```

Here are the commands to do the same things using `tar`:

```
find . -depth -print0 |
  tar --create --null --files-from=- --file=/dev/nrst0

tar --extract --null --preserve-perm --same-owner \
  --file=/dev/nrst0
```

Here is an example of copying a directory from one machine to another:

```
find . -depth -print0 | cpio -0o -Hnewc |
  rsh other-machine "cd `pwd` && cpio -i0dum"
```

7.3 Cleaning Up

This section gives examples of removing unwanted files in various situations. Here is a command to remove the CVS backup files created when an update requires a merge:

```
find . -name '.#*' -print0 | xargs -0r rm -f
```

The command above works, but the following is safer:

```
find . -name '.#*' -depth -delete
```

You can run this command to clean out your clutter in `/tmp`. You might place it in the file your shell runs when you log out (`.bash_logout`, `.logout`, or `.zlogout`, depending on which shell you use).

```
find /tmp -depth -user "$LOGNAME" -type f -delete
```

If your `find` command removes directories, you may find that you get a spurious error message when `find` tries to recurse into a directory that has now been removed. Using the '`-depth`' option will normally resolve this problem.

To remove old Emacs backup and auto-save files, you can use a command like the following. It is especially important in this case to use null-terminated file names because Emacs packages like the VM mailer often create temporary file names with spaces in them, like `#reply to David J. MacKenzie<1>#`.

```
find ~ \( -name '*~' -o -name '#*#' \) -print0 |
  xargs --no-run-if-empty --null rm -vf
```

Removing old files from `/tmp` is commonly done from `cron`:

```
find /tmp /var/tmp -not -type d -mtime +3 -delete
find /tmp /var/tmp -depth -mindepth 1 -type d -empty -delete
```

The second `find` command above uses '`-depth`' so it cleans out empty directories depth-first, hoping that the parents become empty and can be removed too. It uses '`-mindepth`' to avoid removing `/tmp` itself if it becomes totally empty.

7.4 Strange File Names

`find` can help you remove or rename a file with strange characters in its name. People are sometimes stymied by files whose names contain characters such as spaces, tabs, control characters, or characters with the high bit set. The simplest way to remove such files is:

```
rm -i some*pattern*that*matches*the*problem*file
```

`rm` asks you whether to remove each file matching the given pattern. If you are using an old shell, this approach might not work if the file name contains a character with the high bit set; the shell may strip it off. A more reliable way is:

```
find . -maxdepth 1 tests -okdir rm '{}' \;
```

where *tests* uniquely identify the file. The '`-maxdepth 1`' option prevents `find` from wasting time searching for the file in any subdirectories; if there are no subdirectories, you may omit it. A good way to uniquely identify the problem file is to figure out its inode number; use

```
ls -i
```

Suppose you have a file whose name contains control characters, and you have found that its inode number is 12345. This command prompts you for whether to remove it:

```
find . -maxdepth 1 -inum 12345 -okdir rm -f '{}' \;
```

If you don't want to be asked, perhaps because the file name may contain a strange character sequence that will mess up your screen when printed, then use '-execdir' instead of '-okdir'.

If you want to rename the file instead, you can use mv instead of rm:

```
find . -maxdepth 1 -inum 12345 -okdir mv '{}' new-file-name \;
```

7.5 Fixing Permissions

Suppose you want to make sure that everyone can write to the directories in a certain directory tree. Here is a way to find directories lacking either user or group write permission (or both), and fix their permissions:

```
find . -type d -not -perm -ug=w | xargs chmod ug+w
```

You could also reverse the operations, if you want to make sure that directories do *not* have world write permission.

7.6 Classifying Files

If you want to classify a set of files into several groups based on different criteria, you can use the comma operator to perform multiple independent tests on the files. Here is an example:

```
find / -type d \( -perm -o=w -fprint allwrite , \
  -perm -o=x -fprint allexec \)

echo "Directories that can be written to by everyone:"
cat allwrite
echo ""
echo "Directories with search permissions for everyone:"
cat allexec
```

find has only to make one scan through the directory tree (which is one of the most time consuming parts of its work).

8 Worked Examples

The tools in the findutils package, and in particular `find`, have a large number of options. This means that quite often, there is more than one way to do things. Some of the options and facilities only exist for compatibility with other tools, and findutils provides improved ways of doing things.

This chapter describes a number of useful tasks that are commonly performed, and compares the different ways of achieving them.

8.1 Deleting Files

One of the most common tasks that `find` is used for is locating files that can be deleted. This might include:

- Files last modified more than 3 years ago which haven't been accessed for at least 2 years
- Files belonging to a certain user
- Temporary files which are no longer required

This example concentrates on the actual deletion task rather than on sophisticated ways of locating the files that need to be deleted. We'll assume that the files we want to delete are old files underneath `/var/tmp/stuff`.

8.1.1 The Traditional Way

The traditional way to delete files in `var/tmp/stuff` that have not been modified in over 90 days would have been:

```
find /var/tmp/stuff -mtime +90 -exec /bin/rm {} \;
```

The above command uses '`-exec`' to run the `/bin/rm` command to remove each file. This approach works and in fact would have worked in Version 7 Unix in 1979. However, there are a number of problems with this approach.

The most obvious problem with the approach above is that it causes `find` to fork every time it finds a file that needs to delete, and the child process then has to use the `exec` system call to launch `/bin/rm`. All this is quite inefficient. If we are going to use `/bin/rm` to do this job, it is better to make it delete more than one file at a time.

The most obvious way of doing this is to use the shell's command expansion feature:

```
/bin/rm `find /var/tmp/stuff -mtime +90 -print`
```

or you could use the more modern form

```
/bin/rm $(find /var/tmp/stuff -mtime +90 -print)
```

The commands above are much more efficient than the first attempt. However, there is a problem with them. The shell has a maximum command length which is imposed by the operating system (the actual limit varies between systems). This means that while the command expansion technique will usually work, it will suddenly fail when there are lots of files to delete. Since the task is to delete unwanted files, this is precisely the time we don't want things to go wrong.

8.1.2 Making Use of xargs

So, is there a way to be more efficient in the use of `fork()` and `exec()` without running up against this limit? Yes, we can be almost optimally efficient by making use of the `xargs` command. The `xargs` command reads arguments from its standard input and builds them into command lines. We can use it like this:

```
find /var/tmp/stuff -mtime +90 -print | xargs /bin/rm
```

For example if the files found by `find` are `/var/tmp/stuff/A`, `/var/tmp/stuff/B` and `/var/tmp/stuff/C` then `xargs` might issue the commands

```
/bin/rm /var/tmp/stuff/A /var/tmp/stuff/B
/bin/rm /var/tmp/stuff/C
```

The above assumes that `xargs` has a very small maximum command line length. The real limit is much larger but the idea is that `xargs` will run `/bin/rm` as many times as necessary to get the job done, given the limits on command line length.

This usage of `xargs` is pretty efficient, and the `xargs` command is widely implemented (all modern versions of Unix offer it). So far then, the news is all good. However, there is bad news too.

8.1.3 Unusual characters in filenames

Unix-like systems allow any characters to appear in file names with the exception of the ASCII NUL character and the backslash. Backslashes can occur in path names (as the directory separator) but not in the names of actual directory entries. This means that the list of files that `xargs` reads could in fact contain white space characters — spaces, tabs and newline characters. Since by default, `xargs` assumes that the list of files it is reading uses white space as an argument separator, it cannot correctly handle the case where a filename actually includes white space. This makes the default behaviour of `xargs` almost useless for handling arbitrary data.

To solve this problem, GNU findutils introduced the '`-print0`' action for `find`. This uses the ASCII NUL character to separate the entries in the file list that it produces. This is the ideal choice of separator since it is the only character that cannot appear within a path name. The '`-0`' option to `xargs` makes it assume that arguments are separated with ASCII NUL instead of white space. It also turns off another misfeature in the default behaviour of `xargs`, which is that it pays attention to quote characters in its input. Some versions of `xargs` also terminate when they see a lone '`_`' in the input, but GNU `find` no longer does that (since it has become an optional behaviour in the Unix standard).

So, putting `find -print0` together with `xargs -0` we get this command:

```
find /var/tmp/stuff -mtime +90 -print0 | xargs -0 /bin/rm
```

The result is an efficient way of proceeding that correctly handles all the possible characters that could appear in the list of files to delete. This is good news. However, there is, as I'm sure you're expecting, also more bad news. The problem is that this is not a portable construct; although other versions of Unix (notable BSD-derived ones) support '`-print0`', it's not universal. So, is there a more universal mechanism?

8.1.4 Going back to -exec

There is indeed a more universal mechanism, which is a slight modification to the '`-exec`' action. The normal '`-exec`' action assumes that the command to run is terminated with a semicolon (the semicolon normally has to be quoted in order to protect it from interpretation as the shell command separator). The SVR4 edition of Unix introduced a slight variation, which involves terminating the command with '+' instead:

```
find /var/tmp/stuff -mtime +90 -exec /bin/rm {} \+
```

The above use of '-exec' causes `find` to build up a long command line and then issue it. This can be less efficient than some uses of `xargs`; for example `xargs` allows new command lines to be built up while the previous command is still executing, and allows you to specify a number of commands to run in parallel. However, the `find ... -exec ... +` construct has the advantage of wide portability. GNU findutils did not support '-exec ... +' until version 4.2.12; one of the reasons for this is that it already had the '-print0' action in any case.

8.1.5 A more secure version of -exec

The command above seems to be efficient and portable. However, within it lurks a security problem. The problem is shared with all the commands we've tried in this worked example so far, too. The security problem is a race condition; that is, if it is possible for somebody to manipulate the filesystem that you are searching while you are searching it, it is possible for them to persuade your `find` command to cause the deletion of a file that you can delete but they normally cannot.

The problem occurs because the '-exec' action is defined by the POSIX standard to invoke its command with the same working directory as `find` had when it was started. This means that the arguments which replace the {} include a relative path from `find`'s starting point down the file that needs to be deleted. For example,

```
find /var/tmp/stuff -mtime +90 -exec /bin/rm {} \+
```

might actually issue the command:

```
/bin/rm /var/tmp/stuff/A /var/tmp/stuff/B /var/tmp/stuff/passwd
```

Notice the file `/var/tmp/stuff/passwd`. Likewise, the command:

```
cd /var/tmp && find stuff -mtime +90 -exec /bin/rm {} \+
```

might actually issue the command:

```
/bin/rm stuff/A stuff/B stuff/passwd
```

If an attacker can rename `stuff` to something else (making use of their write permissions in `/var/tmp`) they can replace it with a symbolic link to `/etc`. That means that the `/bin/rm` command will be invoked on `/etc/passwd`. If you are running your `find` command as root, the attacker has just managed to delete a vital file. All they needed to do to achieve this was replace a subdirectory with a symbolic link at the vital moment.

There is however, a simple solution to the problem. This is an action which works a lot like `-exec` but doesn't need to traverse a chain of directories to reach the file that it needs to work on. This is the '-execdir' action, which was introduced by the BSD family of operating systems. The command,

```
find /var/tmp/stuff -mtime +90 -execdir /bin/rm {} \+
```

might delete a set of files by performing these actions:

1. Change directory to /var/tmp/stuff/foo
2. Invoke `/bin/rm ./file1 ./file2 ./file3`
3. Change directory to /var/tmp/stuff/bar

4. Invoke `/bin/rm ./file99 ./file100 ./file101`

This is a much more secure method. We are no longer exposed to a race condition. For many typical uses of `find`, this is the best strategy. It's reasonably efficient, but the length of the command line is limited not just by the operating system limits, but also by how many files we actually need to delete from each directory.

Is it possible to do any better? In the case of general file processing, no. However, in the specific case of deleting files it is indeed possible to do better.

8.1.6 Using the -delete action

The most efficient and secure method of solving this problem is to use the '`-delete`' action:

```
find /var/tmp/stuff -mtime +90 -delete
```

This alternative is more efficient than any of the '`-exec`' or '`-execdir`' actions, since it entirely avoids the overhead of forking a new process and using `exec` to run `/bin/rm`. It is also normally more efficient than `xargs` for the same reason. The file deletion is performed from the directory containing the entry to be deleted, so the '`-delete`' action has the same security advantages as the '`-execdir`' action has.

The '`-delete`' action was introduced by the BSD family of operating systems.

8.1.7 Improving things still further

Is it possible to improve things still further? Not without either modifying the system library to the operating system or having more specific knowledge of the layout of the filesystem and disk I/O subsystem, or both.

The `find` command traverses the filesystem, reading directories. It then issues a separate system call for each file to be deleted. If we could modify the operating system, there are potential gains that could be made:

- We could have a system call to which we pass more than one filename for deletion
- Alternatively, we could pass in a list of inode numbers (on GNU/Linux systems, `readdir()` also returns the inode number of each directory entry) to be deleted.

The above possibilities sound interesting, but from the kernel's point of view it is difficult to enforce standard Unix access controls for such processing by inode number. Such a facility would probably need to be restricted to the superuser.

Another way of improving performance would be to increase the parallelism of the process. For example if the directory hierarchy we are searching is actually spread across a number of disks, we might somehow be able to arrange for `find` to process each disk in parallel. In practice GNU `find` doesn't have such an intimate understanding of the system's filesystem layout and disk I/O subsystem.

However, since the system administrator can have such an understanding they can take advantage of it like so:

```
find /var/tmp/stuff1 -mtime +90 -delete &
find /var/tmp/stuff2 -mtime +90 -delete &
find /var/tmp/stuff3 -mtime +90 -delete &
find /var/tmp/stuff4 -mtime +90 -delete &
wait
```

In the example above, four separate instances of `find` are used to search four subdirectories in parallel. The `wait` command simply waits for all of these to complete. Whether this approach is more or less efficient than a single instance of `find` depends on a number of things:

- Are the directories being searched in parallel actually on separate disks? If not, this parallel search might just result in a lot of disk head movement and so the speed might even be slower.
- Other activity - are other programs also doing things on those disks?

8.1.8 Conclusion

The fastest and most secure way to delete files with the help of `find` is to use '`-delete`'. Using `xargs -0 -P N` can also make effective use of the disk, but it is not as secure.

In the case where we're doing things other than deleting files, the most secure alternative is '`-execdir ... +`', but this is not as portable as the insecure action '`-exec ... +`'.

The '`-delete`' action is not completely portable, but the only other possibility which is as secure ('`-execdir`') is no more portable. The most efficient portable alternative is '`-exec ...+`', but this is insecure and isn't supported by versions of GNU findutils prior to 4.2.12.

8.2 Updating A Timestamp File

Suppose we have a directory full of files which is maintained with a set of automated tools; perhaps one set of tools updates them and another set of tools uses the result. In this situation, it might be useful for the second set of tools to know if the files have recently been changed. It might be useful, for example, to have a 'timestamp' file which gives the timestamp on the newest file in the collection.

We can use `find` to achieve this, but there are several different ways to do it.

8.2.1 Updating the Timestamp The Wrong Way

The obvious but wrong answer is just to use '`-newer`':-

```
find subdir -newer timestamp -exec touch -r {} timestamp \;
```

This does the right sort of thing but has a bug. Suppose that two files in the subdirectory have been updated, and that these are called `file1` and `file2`. The command above will update `timestamp` with the modification time of `file1` or that of `file2`, but we don't know which one. Since the timestamps on `file1` and `file2` will in general be different, this could well be the wrong value.

One solution to this problem is to modify `find` to recheck the modification time of `timestamp` every time a file is to be compared against it, but that will reduce the performance of `find`.

8.2.2 Using the test utility to compare timestamps

The `test` command can be used to compare timestamps:

```
find subdir -exec test {} -nt timestamp \; -exec touch -r {} timestamp \;
```

This will ensure that any changes made to the modification time of `timestamp` that take place during the execution of `find` are taken into account. This resolves our earlier problem, but unfortunately this runs much more slowly.

8.2.3 A combined approach

We can of course still use '`-newer`' to cut down on the number of calls to `test`:

```
find subdir -newer timestamp -a \
     -exec test {} -nt timestamp \; -a \
     -exec touch -r {} timestamp \;
```

Here, the '`-newer`' test excludes all the files which are definitely older than the timestamp, but all the files which are newer than the old value of the timestamp are compared against the current updated timestamp.

This is indeed faster in general, but the speed difference will depend on how many updated files there are.

8.2.4 Using -printf and sort to compare timestamps

It is possible to use the '`-printf`' action to abandon the use of `test` entirely:

```
newest=$(find subdir -newer timestamp -printf "%A%p\n" |
           sort -n |
           tail -1 |
           cut -d: -f2- )
touch -r "${newest:-timestamp}" timestamp
```

The command above works by generating a list of the timestamps and names of all the files which are newer than the timestamp. The `sort`, `tail` and `cut` commands simply pull out the name of the file with the largest timestamp value (that is, the latest file). The `touch` command is then used to update the timestamp,

The `"${newest:-timestamp}"` expression simply expands to the value of `$newest` if that variable is set, but to `timestamp` otherwise. This ensures that an argument is always given to the '`-r`' option of the `touch` command.

This approach seems quite efficient, but unfortunately it has a bug. Many operating systems now keep file modification time information at a granularity which is finer than one second. Unfortunately the '`%A@`' format for '`-printf`' only prints a whole-number value currently; that is, these values are at a one-second granularity. This means that in our example above, '`$newest`' will be the name of a file which is no more than one second older than the newest file, but may indeed be older.

It would be possible to solve this problem with some kind of loop:

```
while true; do
        newest=$(find subdir -newer timestamp -printf "%A@:%p\n" |
           sort -n |
           tail -1 |
           cut -d: -f2- )
        if test -z "$newest" ; then
                break
        else
                touch -r "$newest" timestamp
        fi
done
```

A better fix for this problem would be to allow the '`%A@`' format to produce a result having a fractional part, too. While this is planned for GNU `find`, it hasn't been done yet.

8.2.5 Coping with sub-second timestamp resolution

Another tool which often works with timestamps is `make`. We can use `find` to generate a `Makefile` file on the fly and then use `make` to update the timestamps:

```
makefile=$(mktemp)
find subdir \
\( \! -xtype l \) \
-newer timestamp \
-printf "timestamp:: %p\n\ttouch -r %p timestamp\n\n" > "$makefile"
make -f "$makefile"
rm   -f "$makefile"
```

Unfortunately although the solution above is quite elegant, it fails to cope with white space within file names, and adjusting it to do so would require a rather complex shell script.

8.2.6 Coping with odd filenames too

We can fix both of these problems (looping and problems with white space), and do things more efficiently too. The following command works with newlines and doesn't need to sort the list of filenames.

```
find subdir -newer timestamp -printf "%A@:%p\0" |
   perl -0 newest.pl |
   xargs --no-run-if-empty --null -i \
      find {} -maxdepth 0 -newer timestamp -exec touch -r {} timestamp \;
```

The first `find` command generates a list of files which are newer than the original timestamp file, and prints a list of them with their timestamps. The `newest.pl` script simply filters out all the filenames which have timestamps which are older than whatever the newest file is:-

```
#! /usr/bin/perl -0
my @newest = ();
my $latest_stamp = undef;
while (<>) {
    my ($stamp, $name) = split(/:/);
    if (!defined($latest_stamp) || ($tstamp > $latest_stamp)) {
        $latest_stamp = $stamp;
        @newest = ();
    }
    if ($tstamp >= $latest_stamp) {
        push @newest, $name;
    }
}
print join("\0", @newest);
```

This prints a list of zero or more files, all of which are newer than the original timestamp file, and which have the same timestamp as each other, to the nearest second. The second `find` command takes each resulting file one at a time, and if that is newer than the timestamp file, the timestamp is updated.

9 Security Considerations

Security considerations are important if you are using `find` or `xargs` to search for or process files that don't belong to you or which other people have control. Security considerations relating to `locate` may also apply if you have files which you do not want others to see.

The most severe forms of security problems affecting `find` and related programs are when third parties bring about a situation allowing them to do something they would normally not be able to accomplish. This is called *privilege elevation*. This might include deleting files they would not normally be able to delete. It is common for the operating system to periodically invoke `find` for self-maintenance purposes. These invocations of `find` are particularly problematic from a security point of view as these are often invoked by the superuser and search the entire filesystem hierarchy. Generally, the severity of any associated problem depends on what the system is going to do with the files found by `find`.

9.1 Levels of Risk

There are some security risks inherent in the use of `find`, `xargs` and (to a lesser extent) `locate`. The severity of these risks depends on what sort of system you are using:

High risk Multi-user systems where you do not control (or trust) the other users, and on which you execute `find`, including areas where those other users can manipulate the filesystem (for example beneath `/home` or `/tmp`).

Medium Risk
Systems where the actions of other users can create file names chosen by them, but to which they don't have access while `find` is being run. This access might include leaving programs running (shell background jobs, `at` or `cron` tasks, for example). On these sorts of systems, carefully written commands (avoiding use of '`-print`' for example) should not expose you to a high degree of risk. Most systems fall into this category.

Low Risk Systems to which untrusted parties do not have access, cannot create file names of their own choice (even remotely) and which contain no security flaws which might enable an untrusted third party to gain access. Most systems do not fall into this category because there are many ways in which external parties can affect the names of files that are created on your system. The system on which I am writing this for example automatically downloads software updates from the Internet; the names of the files in which these updates exist are chosen by third parties[1].

In the discussion above, "risk" denotes the likelihood that someone can cause `find`, `xargs`, `locate` or some other program which is controlled by them to do something you did not intend. The levels of risk suggested do not take any account of the consequences of this sort of event. That is, if you operate a "low risk" type system, but the consequences of a security problem are disastrous, then you should still give serious thought to all the possible security problems, many of which of course will not be discussed here – this section of the manual is intended to be informative but not comprehensive or exhaustive.

---

[1] Of course, I trust these parties to a large extent anyway, because I install software provided by them; I choose to trust them in this way, and that's a deliberate choice

If you are responsible for the operation of a system where the consequences of a security problem could be very important, you should do two things:-

1. Define a security policy which defines who is allowed to do what on your system.
2. Seek competent advice on how to enforce your policy, detect breaches of that policy, and take account of any potential problems that might fall outside the scope of your policy.

9.2 Security Considerations for `find`

Some of the actions `find` might take have a direct effect; these include `-exec` and `-delete`. However, it is also common to use `-print` explicitly or implicitly, and so if `find` produces the wrong list of file names, that can also be a security problem; consider the case for example where `find` is producing a list of files to be deleted.

We normally assume that the `find` command line expresses the file selection criteria and actions that the user had in mind – that is, the command line is "trusted" data.

From a security analysis point of view, the output of `find` should be correct; that is, the output should contain only the names of those files which meet the user's criteria specified on the command line. This applies for the `-exec` and `-delete` actions; one can consider these to be part of the output.

On the other hand, the contents of the filesystem can be manipulated by other people, and hence we regard this as "untrusted" data. This implies that the `find` command line is a filter which converts the untrusted contents of the filesystem into a correct list of output files.

The filesystem will in general change while `find` is searching it; in fact, most of the potential security problems with `find` relate to this issue in some way.

Race conditions are a general class of security problem where the relative ordering of actions taken by `find` (for example) and something else are critically important in getting the correct and expected result[2] .

For `find`, an attacker might move or rename files or directories in the hope that an action might be taken against a file which was not normally intended to be affected. Alternatively, this sort of attack might be intended to persuade `find` to search part of the filesystem which would not normally be included in the search (defeating the `-prune` action for example).

9.2.1 Problems with -exec and filenames

It is safe in many cases to use the '`-execdir`' action with any file name. Because '`-execdir`' prefixes the arguments it passes to programs with '`./`', you will not accidentally pass an argument which is interpreted as an option. For example the file `-f` would be passed to `rm` as `./-f`, which is harmless.

However, your degree of safety does depend on the nature of the program you are running. For example constructs such as these two commands

```
find -exec sh -c "something {}" \;
find -execdir sh -c "something {}" \;
```

[2] This is more or less the definition of the term "race condition"

are very dangerous. The reason for this is that the '{}' is expanded to a filename which might contain a semicolon or other characters special to the shell. If for example someone creates the file `/tmp/foo; rm -rf $HOME` then the two commands above could delete someone's home directory.

So for this reason do not run any command which will pass untrusted data (such as the names of files) to commands which interpret arguments as commands to be further interpreted (for example 'sh').

9.2.2 Changing the Current Working Directory

As `find` searches the filesystem, it finds subdirectories and then searches within them by changing its working directory. First, `find` reaches and recognises a subdirectory. It then decides if that subdirectory meets the criteria for being searched; that is, any '-xdev' or '-prune' expressions are taken into account. The `find` program will then change working directory and proceed to search the directory.

A race condition attack might take the form that once the checks relevant to '-xdev' and '-prune' have been done, an attacker might rename the directory that was being considered, and put in its place a symbolic link that actually points somewhere else.

The idea behind this attack is to fool `find` into going into the wrong directory. This would leave `find` with a working directory chosen by an attacker, bypassing any protection apparently provided by '-xdev' and '-prune', and any protection provided by being able to *not* list particular directories on the `find` command line. This form of attack is particularly problematic if the attacker can predict when the `find` command will be run, as is the case with `cron` tasks for example.

GNU `find` has specific safeguards to prevent this general class of problem. The exact form of these safeguards depends on the properties of your system.

9.2.2.1 O_NOFOLLOW

If your system supports the O_NOFOLLOW flag[3] to the `open(2)` system call, `find` uses it when safely changing directory. The target directory is first opened and then `find` changes working directory with the `fchdir()` system call. This ensures that symbolic links are not followed, preventing the sort of race condition attack in which use is made of symbolic links.

If for any reason this approach does not work, `find` will fall back on the method which is normally used if O_NOFOLLOW is not supported.

You can tell if your system supports O_NOFOLLOW by running

```
find --version
```

This will tell you the version number and which features are enabled. For example, if I run this on my system now, this gives:

```
GNU find version 4.2.18-CVS
Features enabled: D_TYPE O_NOFOLLOW(enabled)
```

Here, you can see that I am running a version of `find` which was built from the development (CVS) code prior to the release of findutils-4.2.18, and that the D_TYPE and O_NOFOLLOW features are present. O_NOFOLLOW is qualified with "enabled".

[3] GNU/Linux (kernel version 2.1.126 and later) and FreeBSD (3.0-CURRENT and later) support this

This simply means that the current system seems to support O_NOFOLLOW. This check is needed because it is possible to build `find` on a system that defines O_NOFOLLOW and then run it on a system that ignores the O_NOFOLLOW flag. We try to detect such cases at startup by checking the operating system and version number; when this happens you will see "O_NOFOLLOW(disabled)" instead.

9.2.2.2 Systems without O_NOFOLLOW

The strategy for preventing this type of problem on systems that lack support for the O_NOFOLLOW flag is more complex. Each time `find` changes directory, it examines the directory it is about to move to, issues the `chdir()` system call, and then checks that it has ended up in the subdirectory it expected. If all is as expected, processing continues as normal. However, there are two main reasons why the directory might change: the use of an automounter and the someone removing the old directory and replacing it with something else while `find` is trying to descend into it.

Where a filesystem "automounter" is in use it can be the case that the use of the `chdir()` system call can itself cause a new filesystem to be mounted at that point. On systems that do not support O_NOFOLLOW, this will cause `find`'s security check to fail.

However, this does not normally represent a security problem, since the automounter configuration is normally set up by the system administrator. Therefore, if the `chdir()` sanity check fails, `find` will make one more attempt. If that succeeds, execution carries on as normal. This is the usual case for automounters.

Where an attacker is trying to exploit a race condition, the problem may not have gone away on the second attempt. If this is the case, `find` will issue a warning message and then ignore that subdirectory. When this happens, actions such as '`-exec`' or '`-print`' may already have taken place for the problematic subdirectory. This is because `find` applies tests and actions to directories before searching within them (unless '`-depth`' was specified).

Because of the nature of the directory-change operation and security check, in the worst case the only things that `find` would have done with the directory are to move into it and back out to the original parent. No operations would have been performed within that directory.

9.2.3 Race Conditions with -exec

The '`-exec`' action causes another program to be run. It passes to the program the name of the file which is being considered at the time. The invoked program will typically then perform some action on that file. Once again, there is a race condition which can be exploited here. We shall take as a specific example the command

```
find /tmp -path /tmp/umsp/passwd -exec /bin/rm
```

In this simple example, we are identifying just one file to be deleted and invoking `/bin/rm` to delete it. A problem exists because there is a time gap between the point where `find` decides that it needs to process the '`-exec`' action and the point where the `/bin/rm` command actually issues the `unlink()` system call to delete the file from the filesystem. Within this time period, an attacker can rename the `/tmp/umsp` directory, replacing it with a symbolic link to `/etc`. There is no way for `/bin/rm` to determine that it is working on the same file that `find` had in mind. Once the symbolic link is in place, the attacker has persuaded `find` to cause the deletion of the `/etc/passwd` file, which is not the effect intended by the command which was actually invoked.

One possible defence against this type of attack is to modify the behaviour of '-exec' so that the /bin/rm command is run with the argument ./passwd and a suitable choice of working directory. This would allow the normal sanity check that find performs to protect against this form of attack too. Unfortunately, this strategy cannot be used as the POSIX standard specifies that the current working directory for commands invoked with '-exec' must be the same as the current working directory from which find was invoked. This means that the '-exec' action is inherently insecure and can't be fixed.

GNU find implements a more secure variant of the '-exec' action, '-execdir'. The '-execdir' action ensures that it is not necessary to dereference subdirectories to process target files. The current directory used to invoke programs is the same as the directory in which the file to be processed exists (/tmp/umsp in our example, and only the basename of the file to be processed is passed to the invoked command, with a './' prepended (giving ./passwd in our example).

The '-execdir' action refuses to do anything if the current directory is included in the *$PATH* environment variable. This is necessary because '-execdir' runs programs in the same directory in which it finds files – in general, such a directory might be writable by untrusted users. For similar reasons, '-execdir' does not allow '{}' to appear in the name of the command to be run.

9.2.4 Race Conditions with -print and -print0

The '-print' and '-print0' actions can be used to produce a list of files matching some criteria, which can then be used with some other command, perhaps with xargs. Unfortunately, this means that there is an unavoidable time gap between find deciding that one or more files meet its criteria and the relevant command being executed. For this reason, the '-print' and '-print0' actions are just as insecure as '-exec'.

In fact, since the construction

```
find ...  -print | xargs ...
```

does not cope correctly with newlines or other "white space" in file names, and copes poorly with file names containing quotes, the '-print' action is less secure even than '-print0'.

9.3 Security Considerations for xargs

The description of the race conditions affecting the '-print' action of find shows that xargs cannot be secure if it is possible for an attacker to modify a filesystem after find has started but before xargs has completed all its actions.

However, there are other security issues that exist even if it is not possible for an attacker to have access to the filesystem in real time. Firstly, if it is possible for an attacker to create files with names of their choice on the filesystem, then xargs is insecure unless the '-0' option is used. If a file with the name /home/someuser/foo/bar\n/etc/passwd exists (assume that '\n' stands for a newline character), then find ... -print can be persuaded to print three separate lines:

```
/home/someuser/foo/bar

/etc/passwd
```

If it finds a blank line in the input, xargs will ignore it. Therefore, if some action is to be taken on the basis of this list of files, the /etc/passwd file would be included even if this was not the intent of the person running find. There are circumstances in which an attacker can use this to their advantage. The same consideration applies to file names containing ordinary spaces rather than newlines, except that of course the list of file names will no longer contain an "extra" newline.

This problem is an unavoidable consequence of the default behaviour of the `xargs` command, which is specified by the POSIX standard. The only ways to avoid this problem are either to avoid all use of `xargs` in favour for example of '`find -exec`' or (where available) '`find -execdir`', or to use the '`-0`' option, which ensures that `xargs` considers file names to be separated by ASCII NUL characters rather than whitespace. However, useful as this option is, the POSIX standard does not make it mandatory.

9.4 Security Considerations for `locate`

9.4.1 Race Conditions

It is fairly unusual for the output of `locate` to be fed into another command. However, if this were to be done, this would raise the same set of security issues as the use of '`find ... -print`'. Although the problems relating to whitespace in file names can be resolved by the '`-0`' option of `locate`, this still leaves the race condition problems associated with '`find ... -print0`'. There is no way to avoid these problems in the case of `locate`.

9.4.2 Long File Name Bugs with Old-Format Databases

All versions of `locate` prior to 4.2.31, and also versions 4.3.0 to 4.3.6, have a bug in the way that old-format databases are read. They read file names into a fixed-length 1026 byte buffer, allocated on the heap. This buffer is not extended if file names are too long to fit into the buffer. No range checking on the length of the filename is performed. This could in theory lead to a privilege escalation attack.

On systems using the old database format and affected versions of `locate`, carefully-chosen long file names could in theory allow malicious users to run code of their choice as any user invoking locate.

If remote users can choose the names of files stored on your system, and these files are indexed by `updatedb`, this may be a remote security vulnerability. Findutils version 4.2.31 fixes this problem. The `updatedb`, `bigram` and `code` programs do no appear to be affected.

If you are also using GNU coreutils, you can use the following command to determine the length of the longest file name on a given system:

```
find / -print0 | tr -c '\0' 'x' | tr '\0' '\n' | wc -L
```

Although this problem is significant, the old database format is not the default, and use of the old database format is not common. Most installations and most users will not be affected by this problem.

9.5 Summary

Where untrusted parties can create files on the system, or affect the names of files that are created, all uses for `find`, `locate` and `xargs` have known security problems except the following:

Informational use only
: Uses where the programs are used to prepare lists of file names upon which no further action will ever be taken.

'-delete'
: Use of the '-delete' action with `find` to delete files which meet specified criteria

'-execdir'
: Use of the '-execdir' action with `find` where the `PATH` environment variable contains directories which contain only trusted programs.

10 Error Messages

This section describes some of the error messages sometimes made by `find`, `xargs`, or `locate`, explains them and in some cases provides advice as to what you should do about this.

This manual is written in English. The GNU findutils software features translations of error messages for many languages. For this reason the error messages produced by the programs are made to be as self-explanatory as possible. This approach avoids leaving people to figure out which test an English-language error message corresponds to. Error messages which are self-explanatory will not normally be mentioned in this document. For those messages mentioned in this document, only the English-language version of the message will be listed.

10.1 Error Messages From find

'`invalid predicate '-foo'`'

This means that the `find` command line included something that started with a dash or other special character. The `find` program tried to interpret this as a test, action or option, but didn't recognise it. If it was intended to be a test, check what was specified against the documentation. If, on the other hand, the string is the name of a file which has been expanded from a wildcard (for example because you have a '`*`' on the command line), consider using '`./*`' or just '`.`' instead.

'`unexpected extra predicate`'

This usually happens if you have an extra bracket on the command line (for example '`find . -print \)`').

'`Warning: filesystem /path/foo has recently been mounted`'
'`Warning: filesystem /path/foo has recently been unmounted`'

These messages might appear when `find` moves into a directory and finds that the device number and inode are different to what it expected them to be. If the directory `find` has moved into is on an network filesystem (NFS), it will not issue this message, because `automount` frequently mounts new filesystems on directories as you move into them (that is how it knows you want to use the filesystem). So, if you do see this message, be wary — `automount` may not have been responsible. Consider the possibility that someone else is manipulating the filesystem while `find` is running. Some people might do this in order to mislead `find` or persuade it to look at one set of files when it thought it was looking at another set.

'`/path/foo changed during execution of find (old device number 12345, new device number 6789, filesystem type is <whatever>) [ref XXX]`'

This message is issued when `find` moves into a directory and ends up somewhere it didn't expect to be. This happens in one of two circumstances. Firstly, this happens when `automount` intervenes on a system where `find` doesn't know how to determine what the current set of mounted filesystems is.

Secondly, this can happen when the device number of a directory appears to change during a change of current directory, but `find` is moving up the filesystem hierarchy rather than down into it. In order to prevent `find` wandering off into some unexpected part of the filesystem, we stop it at this point.

'`Don't know how to use getmntent() to read '/etc/mtab'. This is a bug.`'
: This message is issued when a problem similar to the above occurs on a system where `find` doesn't know how to figure out the current list of mount points. Ask for help on bug-findutils@gnu.org.

'`/path/foo/bar changed during execution of find (old inode number 12345, new inode number 67893, filesystem type is <whatever>) [ref XXX]"),`'
: This message is issued when `find` moves into a directory and discovers that the inode number of that directory is different from the inode number that it obtained when it examined the directory previously. This usually means that while `find` was deep in a directory hierarchy doing a time consuming operation, somebody has moved one of the parent directories to another location in the same filesystem. This may or may not have been done maliciously. In any case, `find` stops at this point to avoid traversing parts of the filesystem that it wasn't intended. You can use `ls -li` or `find /path -inum 12345 -o -inum 67893` to find out more about what has happened.

'`sanity check of the fnmatch() library function failed.`'
: Please submit a bug report. You may well be asked questions about your system, and if you compiled the `findutils` code yourself, you should keep your copy of the build tree around. The likely explanation is that your system has a buggy implementation of `fnmatch` that looks enough like the GNU version to fool `configure`, but which doesn't work properly.

'`cannot fork`'
: This normally happens if you use the `-exec` action or something similar (`-ok` and so forth) but the system has run out of free process slots. This is either because the system is very busy and the system has reached its maximum process limit, or because you have a resource limit in place and you've reached it. Check the system for runaway processes (with `ps`, if possible). Some process slots are normally reserved for use by '`root`'.

'`some-program terminated by signal 99`'
: Some program which was launched with `-exec` or similar was killed with a fatal signal. This is just an advisory message.

10.2 Error Messages From xargs

'`environment is too large for exec`'
: This message means that you have so many environment variables set (or such large values for them) that there is no room within the system-imposed limits on program command line argument length to invoke any program. This is an unlikely situation and is more likely result of an attempt to test the limits of `xargs`, or break it. Please try unsetting some environment variables, or exiting the current shell. You can also use '`xargs --show-limits`' to understand the relevant sizes.

'can not fit single argument within argument list size limit'
: You are using the '-I' option and xargs doesn't have enough space to build a command line because it has read a really large item and it doesn't fit. You can probably work around this problem with the '-s' option, but the default size is pretty large. This is a rare situation and is more likely an attempt to test the limits of xargs, or break it. Otherwise, you will need to try to shorten the problematic argument or not use xargs.

'cannot fork'
: See the description of the similar message for find.

'<program>: exited with status 255; aborting'
: When a command run by xargs exits with status 255, xargs is supposed to stop. If this is not what you intended, wrap the program you are trying to invoke in a shell script which doesn't return status 255.

'<program>: terminated by signal 99'
: See the description of the similar message for find.

10.3 Error Messages From locate

'warning: database '/usr/local/var/locatedb' is more than 8 days old'
: The locate program relies on a database which is periodically built by the updatedb program. That hasn't happened in a long time. To fix this problem, run updatedb manually. This can often happen on systems that are generally not left on, so the periodic "cron" task which normally does this doesn't get a chance to run.

'locate database '/usr/local/var/locatedb' is corrupt or invalid'
: This should not happen. Re-run updatedb. If that works, but locate still produces this error, run locate --version and updatedb --version. These should produce the same output. If not, you are using a mixed toolset; check your '$PATH' environment variable and your shell aliases (if you have any). If both programs claim to be GNU versions, this is a bug; all versions of these programs should interoperate without problem. Ask for help on bug-findutils@gnu.org.

10.4 Error Messages From updatedb

The updatedb program (and the programs it invokes) do issue error messages, but none seem to be candidates for guidance. If you are having a problem understanding one of these, ask for help on bug-findutils@gnu.org.

Appendix A GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

   The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

   Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

find Primary Index

This is a list of all of the primaries (tests, actions, and options) that make up find expressions for selecting files. See Section 1.3 [find Expressions], page 3, for more information on expressions.

(Index is nonexistent)