Finding Files

Edition 1.1, for GNU `find` version 4.1
November 1994

by David MacKenzie

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. Many other people have contributed bug fixes, small improvements, and helpful suggestions. Thanks!

Mail suggestions and bug reports for these programs to `bug-gnu-utils@prep.ai.mit.edu`. Please include the version number, which you can get by running '`find --version`'.

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
```

`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 12, 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 13).

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 13, 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 5, 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. To ignore a whole directory tree, use '`-prune`' (see Section 2.9 [Directories], page 10). As an example, to find Texinfo source files in '`/usr/local/doc`':

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

2.1.2 Full Name Patterns

-path *pattern* Test
-ipath *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 '`-ipath`', 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 10).

-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 '`b.*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.

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 5, 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.

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 5).

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. 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.

`*` 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.

In the `find` tests that do shell pattern matching ('`-name`', '`-path`', etc.), wildcards in the pattern do not match a '`.`' at the beginning of a file name. This is not the case for `locate`. Thus, '`find -name '*macs'`' does not match a file named '`.emacs`', but '`locate '*macs'`' does.

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`'.

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

-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 5, for details about the *pattern* argument. So, to list any symbolic links to '`sysdep.c`' in the current directory and its subdirectories, you can do:

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

-follow Option

Dereference symbolic links. The following differences in behavior occur when this option is given:

- `find` follows symbolic links to directories when searching directory trees.
- '`-lname`' and '`-ilname`' always return false.
- '`-type`' reports the types of the files that symbolic links point to.
- Implies '`-noleaf`' (see Section 2.9 [Directories], page 10).

2.2.2 Hard Links

To find hard links, first get the inode number of the file whose links you want to find. You can learn a file's inode number and the number of links to it by running '`ls -i`' or '`find -ls`'. If the file has more than one link, you can search for the other links by passing that inode number to '`-inum`'. Add the '`-xdev`' option if you are starting the search at a directory that has other filesystems mounted on it, such as '`/usr`' on many systems. Doing this saves needless searching, since hard links to a file must be on the same filesystem. See Section 2.10 [Filesystems], page 11.

-inum *n* — Test

File has inode number *n*.

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.

-links *n* — Test

File has *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)

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.

-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. For example, to list files in '`/u/bill`' that were last read from 2 to 6 hours 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
```

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 6, 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*[*bckw*] 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

`c` bytes

`k` kilobytes (1024 bytes)

`w` 2-byte words

The size does not count indirect blocks, but it does count blocks in sparse files that are not actually allocated.

-empty Test

True if the file is empty and is either a regular file or a directory. This might make it a good candidate for deletion. This test is useful with '`-depth`' (see Section 2.9 [Directories], page 10) and '`-exec rm -rf '{}' ';'`' (see Section 3.3.1 [Single File], page 17).

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

`s` socket

-xtype *c* Test

The same as '`-type`' unless the file is a symbolic link. For symbolic links: if '`-follow`' has not been given, true if the file is a link to a file of type *c*; if '`-follow`' has been given, true if *c* is '`l`'. In other words, for symbolic links, '`-xtype`' checks the type of the file that '`-type`' does not check. See Section 2.2.1 [Symbolic Links], page 6, for more information on '`-follow`'.

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 6 [File Permissions], page 28, 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). Symbolic modes use mode 0 as a point of departure. 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.

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.

-prune Action

If '`-depth`' is not given, true; do not descend the current directory. If '`-depth`' is given, false; no effect. '`-prune`' only affects tests and actions that come after it in the expression, not those that come before.

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 . -path './src/emacs' -prune -o -print
```

-noleaf Option

Do not optimize by assuming that directories contain 2 fewer subdirectories than their hard link count. This option is 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.

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:

```
ufs 4.2 4.3 nfs tmp mfs S51K S52K
```

You can use '-printf' with the '%F' directive to see the types of your filesystems. See Section 3.2 [Print File Information], page 13. '-fstype' is usually used with '-prune' to avoid searching remote filesystems (see Section 2.9 [Directories], page 10).

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.

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.

`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 full file name on the standard output, followed by a newline.

-fprint *file* Action

True; print the full 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 file names '`/dev/stdout`' and '`/dev/stderr`' are handled specially; they refer to the standard output and standard error output, respectively.

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 7, 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 8, 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 9). 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 6 [File Permissions], page 28, for more details about file permissions. See Section 2.7 [Permissions], page 10, 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.

-fls *file* Action

True; like '`-ls`' but write to *file* like '`-fprint`' (see Section 3.1 [Print File Name], page 13).

-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. Unlike '`-print`', '`-printf`' does not add a newline at the end of the string.

-fprintf *file format* Action

True; like '`-printf`' but write to *file* like '`-fprint`' (see Section 3.1 [Print File Name], page 13).

3.2.1 Escapes

The escapes that '`-printf`' and '`-fprintf`' recognize 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 ('`\`').

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. Unlike the C `printf` function, they do not support field width specifiers.

'`%%`' is a literal percent sign. A '`%`' character followed by any other character is discarded (but the other character is printed), and a warning message is printed to the standard error output (because it was probably a typo).

3.2.2.1 Name Directives

`%p` File's name.

`%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).

`%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).

3.2.2.3 Size Directives

`%k` File's size in 1K blocks (rounded up).

`%b` File's size in 512-byte blocks (rounded up).

`%s` File's size in bytes.

3.2.2.4 Location Directives

`%d` File's depth in the directory tree; files named on the command line have a depth of 0.

`%F` Type of the filesystem the file is on; this value can be used for '`-fstype`' (see Section 2.9 [Directories], page 10).

`%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.

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 16).

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

`%C`*k* File's last status change time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 16).

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

`%T`*k* File's last modification time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 16).

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)

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.

-exec *command ;* Action

Execute *command*; true if 0 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 the current directory with the file '`/tmp/master`':

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

3.3.2 Multiple Files

Sometimes you need to process files alone. But when you don't, 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.

To run a command on more than one file at once, use the `xargs` command, which is invoked like this:

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

`xargs` reads arguments from the standard input, delimited by blanks (which can be protected with double or single quotes or a backslash) or newlines. It executes the *command* (default is '`/bin/echo`') 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`.

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 | 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.

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 full 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 13).

3.3.2.3 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 20k, 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.

`--max-lines`[=*max-lines*]
`-l`[*max-lines*]
: Use at most *max-lines* nonblank input lines per command line; *max-lines* defaults to 1 if omitted. Trailing blanks cause an input line to be logically continued on the next input line, for the purpose of counting the lines. Implies '`-x`'.

`--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.

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

`--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.

3.3.2.4 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 3.3.2.3 [Limiting Command Size], page 19), this mode of operation is equivalent to '`find -exec`' (see Section 3.3.1 [Single File], page 17).

`--replace`[`=`*replace-str*]
`-i`[*replace-str*]
: Replace occurences of *replace-str* in the initial arguments with names read from standard input. Also, unquoted blanks do not terminate arguments. If *replace-str* is omitted, it defaults to '`{}`' (like for '`find -exec`'). Implies '`-x`' and '`-l 1`'. As an example, to sort each file the '`bills`' directory, leaving the output in that file name with '`.sorted`' appended, you could do:

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

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

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

3.3.3 Querying

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

-ok *command ;* Action
: Like '`-exec`' (see Section 3.3.1 [Single File], page 17), 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.

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 3.3.2.3 [Limiting Command Size], page 19).

`--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 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 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`.

If you want to place a special test somewhere in the middle of a `find` expression, you can use '`-exec`' to run a program that performs the test. Because '`-exec`' evaluates to the exit status of the executed program, you can write 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. Use this method only when `xargs` is not flexible enough, because starting one or more new processes to test each file is slower than using `xargs` to start one process that tests many files.

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

```
#!/bin/sh
file $1 | grep 'not stripped' > /dev/null
```

This script relies on the fact that the shell exits with the status of the last program it executed, in this case `grep`. `grep` 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 in the file '`sbins`' and the unstripped ones in '`ubins`'.

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

4 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.

4.1 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.

4.2 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"
```

4.3 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
```

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 -user $LOGNAME -type f -print0 | xargs -0 -r rm -f
```

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 -print0 |
  xargs --null --no-run-if-empty rm -f

find /tmp /var/tmp -depth -mindepth 1 -type d -empty -print0 |
  xargs --null --no-run-if-empty rmdir
```

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.

4.4 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* `-ok 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 -ok 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 '`-exec`' instead of '`-ok`'.

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

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

4.5 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.

4.6 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).

5 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.

5.1 Database Locations

There can be multiple file name databases. Users can select which databases `locate` searches using an 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. Here are the options to `updatedb` to select which directories each database contains entries for:

`--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. Default is none.

`--prunepaths='`*path...*`'`
: Directories to not put in the database, which would otherwise be. Default is '`/tmp /usr/tmp /var/tmp /afs`'.

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

`--netuser=`*user*
: The user to search network directories as, using `su`. Default is `daemon`.

5.2 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 diffent 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.

5.2.1 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 can not 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.

5.2.2 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)

5.2.3 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.

`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.

6 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.

6.1 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 filesystem 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. save the program's text image on the swap device so it will load more quickly when run (called the *sticky bit*). For directories on some systems, prevent users from removing files that they do not own in the directory; this is called making the directory *append-only*.

6.2 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 6.2.6 [Umask and Protection], page 32).

The format of symbolic modes is:

`[ugoa...][[+-=][rwxXstugo...]...][,...]`

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

6.2.1 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 can not 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 6.2.6 [Umask and Protection], page 32, 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 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 from 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-rxw
```

6.2.2 Copying Existing Permissions

You can base part of a file's permissions on part of its existing permissions. To do this, instead of using '`r`', '`w`', or '`x`' 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.

6.2.3 Changing Special Permissions

In addition to changing a file's read, write, and execute permissions, you can change its special permissions. See Section 6.1 [Mode Structure], page 28, 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 stay permanently on the swap device, use '`o`' in the *users* part of the symbolic mode and '`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 cause a program to be saved on the swap device, you can use the mode:

```
o+t
```

Remember that the special permissions only affect files that are executable, plus, on some systems, directories (on which they have different meanings; see Section 6.1 [Mode Structure], page 28). Using '`a`' in the *users* part of a symbolic mode does not cause the special permissions to be affected; thus,

```
a+s
```

has *no effect*. You must use '`u`', '`g`', and '`o`' explicitly to affect the special permissions. Also, the combinations '`u+t`', '`g+t`', and '`o+s`' have no effect.

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

```
o=t
```

does cause the file to be saved on the swap device, but it also removes all read, write, and execute permissions that users not in the file's group might have had for it.

6.2.4 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 already had execute permission or is a directory. It affects directories' execute permission even if they did not initially have any execute permissions set.

For example, this mode:

```
a+X
```

gives all users permission to execute files (or search directories) if anyone could before.

6.2.5 Making Multiple Changes

The format of symbolic modes is actually more complex than described above (see Section 6.2.1 [Setting Permissions], page 29). 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 it 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
```

6.2.6 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.

6.3 Numeric Modes

File permissions are stored internally as 16 bit integers. As an alternative to giving a symbolic mode, you can give an octal (base 8) number that corresponds to the internal representation of 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 can not 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 are each stored as three bits, which are represented as one octal digit. The three special permissions are also each stored as one bit, and they are as a group represented as another octal digit. Here is how the bits are arranged in the 16 bit integer, 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      Save text image on swap device
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 '`ugo=`'.

7 Reference

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

7.1 Invoking `find`

`find` [*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.

`find` considers the first argument that begins with '`-`', '`(`', '`)`', '`,`', or '`!`' to be the beginning of the expression; any arguments before it are paths to search, and any arguments after it are the rest of the expression. If no paths are given, the current directory is used. If no expression is given, the expression '`-print`' is used.

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

See [Primary Index], page 37, for a summary of all of the tests, actions, and options that the expression can contain.

`find` also recognizes two options for administrative use:

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

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

7.2 Invoking `locate`

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

`--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. 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.

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

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

7.3 Invoking `updatedb`

`updatedb` [*option*...]

`--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. Default is none.

`--prunepaths='`*path...*`'`
: Directories to not put in the database, which would otherwise be. Default is '`/tmp /usr/tmp /var/tmp /afs`'.

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

`--netuser=`*user*
: The user to search network directories as, using `su`(1). Default is `daemon`.

7.4 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.

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

`--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, there is no end of file string. If this option is not given, the end of file string defaults to '`_`'.

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

`--replace`[`=`*replace-str*]
`-i`[*replace-str*]
: Replace occurences of *replace-str* in the initial arguments with names read from standard input. Also, unquoted blanks do not terminate arguments. If *replace-str* is omitted, it defaults to '`{}`' (like for '`find -exec`'). Implies '`-x`' and '`-l 1`'.

`--max-lines`[`=`*max-lines*]
`-l`[*max-lines*]
: Use at most *max-lines* nonblank input lines per command line; *max-lines* defaults to 1 if omitted. Trailing blanks cause an input line to be logically continued on the next input line, for the purpose of counting the lines. Implies '`-x`'.

`--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 does not contain any nonblanks, 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 and initial arguments and the terminating nulls at the ends of the argument strings.

`--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 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.

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 2, for more information on expressions.

Table of Contents