# GNU fileutils

A set of file utilities
for version 4.1, 23 April 2001

**David MacKenzie et al.**

# 1 Introduction

This manual is incomplete: No attempt is made to explain basic file concepts in a way suitable for novices. Thus, if you are interested, please get involved in improving this manual. The entire GNU community will benefit.

The GNU file utilities are mostly compatible with the POSIX.2 standard.

Please report bugs to `bug-fileutils@gnu.org`. Remember to include the version number, machine architecture, input files, and any other information needed to reproduce the bug: your input, what you expected, what you got, and why it is wrong. Diffs are welcome, but please include a description of the problem as well, since this is sometimes difficult to infer. See section "Bugs" in *GNU CC*.

This manual was originally derived from the Unix man pages in the distribution, which were written by David MacKenzie and updated by Jim Meyering. What you are reading now is the authoritative documentation for these utilities; the man pages are no longer being maintained. François Pinard did the initial conversion to Texinfo format. Karl Berry did the indexing, some reorganization, and editing of the results. Richard Stallman contributed his usual invaluable insights to the overall process.

# 2 Common options

Certain options are available in all of these programs (in fact, every GNU program should accept them). Rather than writing identical descriptions for each of the programs, they are described here.

'--help' Print a usage message listing all available options, then exit successfully.

'--version'
: Print the version number, then exit successfully.

## 2.1 Backup options

Some GNU programs (at least `cp`, `install`, `ln`, and `mv`) optionally make backups of files before writing new versions. These options control the details of these backups. The options are also briefly mentioned in the descriptions of the particular programs.

'-b'
'--backup[=*method*]'
: Make a backup of each file that would otherwise be overwritten or removed. Without this option, the original versions are destroyed. Use *method* to determine the type of backups to make. When this option is used but *method* is not specified, then the value of the `VERSION_CONTROL` environment variable is used. And if `VERSION_CONTROL` is not set, the default backup type is '`existing`'.

  Note that the short form of this option, '`-b`' does not accept any argument. Using '`-b`' is equivalent to using '`--backup=existing`'.

  This option corresponds to the Emacs variable '`version-control`'; the values for *method* are the same as those used in Emacs. This option also accepts more descriptive names. The valid *methods* are (unique abbreviations are accepted):

  '`none`'
  '`off`' Never make backups.

  '`numbered`'
  '`t`' Always make numbered backups.

  '`existing`'
  '`nil`' Make numbered backups of files that already have them, simple backups of the others.

  '`simple`'
  '`never`' Always make simple backups. Please note '`never`' is not to be confused with '`none`'.

'-S *suffix*'
'--suffix=*suffix*'
: Append *suffix* to each backup file made with '`-b`'. If this option is not specified, the value of the `SIMPLE_BACKUP_SUFFIX` environment variable is used. And if `SIMPLE_BACKUP_SUFFIX` is not set, the default is '`~`', just as in Emacs.

'--version-control=*method*'
: This option is obsolete and will be removed in a future release. It has been replaced with `--backup`.

## 2.2 Block size

Some GNU programs (at least `df`, `du`, and `ls`) display file sizes in "blocks". You can adjust the block size to make file sizes easier to read. The block size used for display is independent of any filesystem block size.

Normally, disk usage sizes are rounded up, disk free space sizes are rounded down, and other sizes are rounded to the nearest value with ties rounding to an even value.

The default block size is chosen by examining the following environment variables in turn; the first one that is set determines the block size.

`DF_BLOCK_SIZE`
: This specifies the default block size for the `df` command. Similarly, `DU_BLOCK_SIZE` specifies the default for `du` and `LS_BLOCK_SIZE` for `ls`.

`BLOCK_SIZE`
: This specifies the default block size for all three commands, if the above command-specific environment variables are not set.

`POSIXLY_CORRECT`
: If neither the *command*`_BLOCK_SIZE` nor the `BLOCK_SIZE` variables are set, but this variable is set, the block size defaults to 512.

If none of the above environment variables are set, the block size currently defaults to 1024 bytes, but this number may change in the future.

A block size specification can be a positive integer specifying the number of bytes per block, or it can be `human-readable` or `si` to select a human-readable format.

With human-readable formats, output sizes are followed by a size letter such as '`M`' for megabytes. `BLOCK_SIZE=human-readable` uses powers of 1024; '`M`' stands for 1,048,576 bytes. `BLOCK_SIZE=si` is similar, but uses powers of 1000; '`M`' stands for 1,000,000 bytes. (SI, the International System of Units, defines these power-of-1000 prefixes.)

An integer block size can be followed by a size letter to specify a multiple of that size. When this notation is used, the size letters normally stand for powers of 1024, and can be followed by an optional '`B`' for "byte"; but if followed by '`D`' (for "decimal byte"), they stand for powers of 1000. For example, `BLOCK_SIZE=4MB` is equivalent to `BLOCK_SIZE=4194304`, and `BLOCK_SIZE=4MD` is equivalent to `BLOCK_SIZE=4000000`.

The following size letters are defined. Large sizes like `1Y` may be rejected by your computer due to limitations of its arithmetic.

'`k`'
: kilo: $2\hat{}10 = 1024$ for `human-readable`, or $10\hat{}3 = 1000$ for `si`.

'`M`'
: Mega: $2\hat{}20 = 1,048,576$ or $10\hat{}6 = 1,000,000$.

'`G`'
: Giga: $2\hat{}30 = 1,073,741,824$ or $10\hat{}9 = 1,000,000,000$.

'`T`'
: Tera: $2\hat{}40 = 1,099,511,627,776$ or $10\hat{}12 = 1,000,000,000,000$.

'`P`'
: Peta: $2\hat{}50 = 1,125,899,906,842,624$ or $10\hat{}15 = 1,000,000,000,000,000$.

'`E`'
: Exa: $2\hat{}60 = 1,152,921,504,606,846,976$ or $10\hat{}18 = 1,000,000,000,000,000,000$.

'`Z`'
: Zetta: $2\hat{}70 = 1,180,591,620,717,411,303,424$ or $10\hat{}21 = 1,000,000,000,000,000,000,000$.

'Y' Yotta: 2^80 = 1,208,925,819,614,629,174,706,176 or 10^24 = 1,000,000,000,000,000,000,000,000.

Block size defaults can be overridden by an explicit '`--block-size=`*size*' option. The '`-k`' or '`--kilobytes`' option is equivalent to '`--block-size=1k`', which is the default unless the `POSIXLY_CORRECT` environment variable is set. The '`-h`' or '`--human-readable`' option is equivalent to '`--block-size=human-readable`'. The '`--si`' option is equivalent to '`--block-size=si`'.

## 2.3 Target directory

Some GNU programs (at least `cp`, `install`, `ln`, and `mv`) allow you to specify the target directory via this option:

'`--target-directory`=*directory*'

Specify the destination *directory*.

The interface for most programs is that after processing options and a finite (possibly zero) number of fixed-position arguments, the remaining argument list is either expected to be empty, or is a list of items (usually files) that will all be handled identically. The `xargs` program is designed to work well with this convention.

The commands in the `mv`-family are unusual in that they take a variable number of arguments with a special case at the *end* (namely, the target directory). This makes it nontrivial to perform some operations, e.g., "move all files from here to ../d/", because `mv * ../d/` might exhaust the argument space, and `ls | xargs ...` doesn't have a clean way to specify an extra final argument for each invocation of the subject command. (It can be done by going through a shell command, but that requires more human labor and brain power than it should.)

The `--target-directory` option allows the `cp`, `install`, `ln`, and `mv` programs to be used conveniently with `xargs`. For example, you can move the files from the current directory to a sibling directory, `d` like this: (However, this doesn't move files whose names begin with '`.`'.)

```
ls |xargs mv --target-directory=../d
```

If you use the GNU `find` program, you can move *all* files with this command:

```
find . -mindepth 1 -maxdepth 1 \
  | xargs mv --target-directory=../d
```

But that will fail if there are no files in the current directory or if any file has a name containing a newline character. The following example removes those limitations and requires both GNU `find` and GNU `xargs`:

```
find . -mindepth 1 -maxdepth 1 -print0 \
  | xargs --null --no-run-if-empty \
      mv --target-directory=../d
```

## 2.4 Trailing slashes

Some GNU programs (at least `cp` and `mv`) allow you to remove any trailing slashes from each *source* argument before operating on it. The `--strip-trailing-slashes` option enables this behavior.

This is useful when a *source* argument may have a trailing slash and specify a symbolic link to a directory. This scenario is in fact rather common because some shells can automatically append a trailing slash when performing file name completion on such symbolic links. Without this option, `mv`, for example, (via the system's rename function) must interpret a trailing slash as a request to dereference the symbolic link and so must rename the indirectly referenced *directory* and not the symbolic link. Although it may seem surprising that such behavior be the default, it is required by POSIX.2 and is consistent with other parts of that standard.

# 3 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.

## 3.1 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 or renaming a file in a directory unless they own the file or the directory; this is called the *restriction deletion flag* for the directory.

## 3.2 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 3.2.6 [Umask and Protection], page 10).

The format of symbolic modes is:

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

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

### 3.2.1 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 3.2.6 [Umask and Protection], page 10, 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
```

### 3.2.2 Copying Existing Permissions

You can base a file's permissions on 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.

### 3.2.3 Changing Special Permissions

In addition to changing a file's read, write, and execute permissions, you can change its special permissions. See Section 3.1 [Mode Structure], page 6, 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 3.1 [Mode Structure], page 6). 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.

### 3.2.4 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.

### 3.2.5 Making Multiple Changes

The format of symbolic modes is actually more complex than described above (see Section 3.2.1 [Setting Permissions], page 7). 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
```

### 3.2.6 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.

## 3.3 Numeric Modes

File permissions are stored internally as 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 cannot take into account a file's previous permissions; it can only set them absolutely.

On most systems, 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, 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=`'.

# 4 Date input formats

First, a quote:

> Our units of temporal measurement, from seconds on up to months, are so complicated, asymmetrical and disjunctive so as to make coherent mental reckoning in time all but impossible. Indeed, had some tyrannical god contrived to enslave our minds to time, to make it all but impossible for us to escape subjection to sodden routines and unpleasant surprises, he could hardly have done better than handing down our present system. It is like a set of trapezoidal building blocks, with no vertical or horizontal surfaces, like a language in which the simplest thought demands ornate constructions, useless particles and lengthy circumlocutions. Unlike the more successful patterns of language and science, which enable us to face experience boldly or at least level-headedly, our system of temporal calculation silently and persistently encourages our terror of time.
>
> . . . It is as though architects had to measure length in feet, width in meters and height in ells; as though basic instruction manuals demanded a knowledge of five different languages. It is no wonder then that we often look into our own immediate past or future, last Tuesday or a week from Sunday, with feelings of helpless confusion. . . .
>
> — Robert Grudin, *Time and the Art of Living*.

This section describes the textual date representations that GNU programs accept. These are the strings you, as a user, can supply as arguments to the various programs. The C interface (via the `getdate` function) is not described here.

Although the date syntax here can represent any possible time since the year zero, computer integers often cannot represent such a wide range of time. On POSIX systems, the clock starts at 1970-01-01 00:00:00 UTC: POSIX does not require support for times before the POSIX Epoch and times far in the future. Traditional Unix systems have 32-bit signed `time_t` and can represent times from 1901-12-13 20:45:52 through 2038-01-19 03:14:07 UTC. Systems with 64-bit signed `time_t` can represent all the times in the known lifetime of the universe.

## 4.1 General date syntax

A *date* is a string, possibly empty, containing many items separated by whitespace. The whitespace may be omitted when no ambiguity arises. The empty string means the beginning of today (i.e., midnight). Order of the items is immaterial. A date string may contain many flavors of items:

- calendar date items
- time of the day items
- time zone items
- day of the week items
- relative items
- pure numbers.

We describe each of these item types in turn, below.

A few numbers may be written out in words in most contexts. This is most useful for specifying day of the week items or relative items (see below). Here is the list: '`first`' for 1, '`next`' for 2, '`third`' for 3, '`fourth`' for 4, '`fifth`' for 5, '`sixth`' for 6, '`seventh`' for 7, '`eighth`' for 8, '`ninth`' for 9, '`tenth`' for 10, '`eleventh`' for 11 and '`twelfth`' for 12. Also, '`last`' means exactly −1.

When a month is written this way, it is still considered to be written numerically, instead of being "spelled in full"; this changes the allowed strings.

In the current implementation, only English is supported for words and abbreviations like '`AM`', '`DST`', '`EST`', '`first`', '`January`', '`Sunday`', '`tomorrow`', and '`year`'.

The output of `date` is not always acceptable as a date string, not only because of the language problem, but also because there is no standard meaning for time zone items like '`IST`'. When using `date` to generate a date string intended to be parsed later, specify a date format that is independent of language and that does not use time zone items other than '`UTC`' and '`Z`'. Here are some ways to do this:

```
$ LC_ALL=C TZ=UTC0 date
Fri Dec 15 19:48:05 UTC 2000
$ TZ=UTC0 date +"%Y-%m-%d %H:%M:%SZ"
2000-12-15 19:48:05Z
$ date --iso-8601=seconds  # a GNU extension
2000-12-15T11:48:05-0800
$ date --rfc-822  # a GNU extension
Fri, 15 Dec 2000 11:48:05 -0800
$ date +"%Y-%m-%d %H:%M:%S %z"  # %z is a GNU extension.
2000-12-15 11:48:05 -0800
```

Alphabetic case is completely ignored in dates. Comments may be introduced between round parentheses, as long as included parentheses are properly nested. Hyphens not followed by a digit are currently ignored. Leading zeros on numbers are ignored.

## 4.2 Calendar date items

A *calendar date item* specifies a day of the year. It is specified differently, depending on whether the month is specified numerically or literally. All these strings specify the same calendar date:

```
1972-09-24     # ISO 8601.
72-9-24        # Assume 19xx for 69 through 99,
               # 20xx for 00 through 68.
72-09-24       # Leading zeros are ignored.
9/24/72        # Common U.S. writing.
24 September 1972
24 Sept 72     # September has a special abbreviation.
24 Sep 72      # Three-letter abbreviations always allowed.
Sep 24, 1972
24-sep-72
24sep72
```

The year can also be omitted. In this case, the last specified year is used, or the current year if none. For example:

```
9/24
sep 24
```

Here are the rules.

For numeric months, the ISO 8601 format '*year*-*month*-*day*' is allowed, where *year* is any positive number, *month* is a number between 01 and 12, and *day* is a number between 01 and 31. A leading zero must be present if a number is less than ten. If *year* is 68 or smaller, then 2000 is added to it; otherwise, if *year* is less than 100, then 1900 is added to it. The construct '*month*/*day*/*year*', popular in the United States, is accepted. Also '*month*/*day*', omitting the year.

Literal months may be spelled out in full: 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November' or 'December'. Literal months may be abbreviated to their first three letters, possibly followed by an abbreviating dot. It is also permitted to write 'Sept' instead of 'September'.

When months are written literally, the calendar date may be given as any of the following:

*day month year*
*day month*
*month day year*
*day*-*month*-*year*

Or, omitting the year:

*month day*

## 4.3 Time of day items

A *time of day item* in date strings specifies the time on a given day. Here are some examples, all of which represent the same time:

```
20:02:0
20:02
8:02pm
20:02-0500      # In EST (U.S. Eastern Standard Time).
```

More generally, the time of the day may be given as '*hour*:*minute*:*second*', where *hour* is a number between 0 and 23, *minute* is a number between 0 and 59, and *second* is a number between 0 and 59. Alternatively, ':*second*' can be omitted, in which case it is taken to be zero.

If the time is followed by 'am' or 'pm' (or 'a.m.' or 'p.m.'), *hour* is restricted to run from 1 to 12, and ':*minute*' may be omitted (taken to be zero). 'am' indicates the first half of the day, 'pm' indicates the second half of the day. In this notation, 12 is the predecessor of 1: midnight is '12am' while noon is '12pm'. (This is the zero-oriented interpretation of '12am' and '12pm', as opposed to the old tradition derived from Latin which uses '12m' for noon and '12pm' for midnight.)

The time may alternatively be followed by a time zone correction, expressed as '*s**hh**mm*', where *s* is '+' or '-', *hh* is a number of zone hours and *mm* is a number of zone minutes. When a time zone correction is given this way, it forces interpretation of the time relative to Coordinated Universal Time (UTC), overriding any previous specification for the time zone or the local time zone. The *minute* part of the time of the day may not be elided when a time zone correction is used. This is the best way to specify a time zone correction by fractional parts of an hour.

Either 'am'/'pm' or a time zone correction may be specified, but not both.

## 4.4 Time zone items

A *time zone item* specifies an international time zone, indicated by a small set of letters, e.g., 'UTC' or 'Z' for Coordinated Universal Time. Any included periods are ignored. By following a non-daylight-saving time zone by the string 'DST' in a separate word (that is, separated by some white space), the corresponding daylight saving time zone may be specified.

Time zone items other than 'UTC' and 'Z' are obsolescent and are not recommended, because they are ambiguous; for example, 'EST' has a different meaning in Australia than in the United States. Instead, it's better to use unambiguous numeric time zone corrections like '-0500', as described in the previous section.

## 4.5 Day of week items

The explicit mention of a day of the week will forward the date (only if necessary) to reach that day of the week in the future.

Days of the week may be spelled out in full: 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' or 'Saturday'. Days may be abbreviated to their first three letters, optionally followed by a period. The special abbreviations 'Tues' for 'Tuesday', 'Wednes' for 'Wednesday' and 'Thur' or 'Thurs' for 'Thursday' are also allowed.

A number may precede a day of the week item to move forward supplementary weeks. It is best used in expression like 'third monday'. In this context, 'last *day*' or 'next *day*' is also acceptable; they move one week before or after the day that *day* by itself would represent.

A comma following a day of the week item is ignored.

## 4.6 Relative items in date strings

*Relative items* adjust a date (or the current date if none) forward or backward. The effects of relative items accumulate. Here are some examples:

```
1 year
1 year ago
3 years
2 days
```

The unit of time displacement may be selected by the string 'year' or 'month' for moving by whole years or months. These are fuzzy units, as years and months are not all of equal duration. More precise units are 'fortnight' which is worth 14 days, 'week' worth 7 days, 'day' worth 24 hours, 'hour' worth 60 minutes, 'minute' or 'min' worth 60 seconds, and 'second' or 'sec' worth one second. An 's' suffix on these units is accepted and ignored.

The unit of time may be preceded by a multiplier, given as an optionally signed number. Unsigned numbers are taken as positively signed. No number at all implies 1 for a multiplier.

Following a relative item by the string '`ago`' is equivalent to preceding the unit by a multiplier with value $-1$.

The string '`tomorrow`' is worth one day in the future (equivalent to '`day`'), the string '`yesterday`' is worth one day in the past (equivalent to '`day ago`').

The strings '`now`' or '`today`' are relative items corresponding to zero-valued time displacement, these strings come from the fact a zero-valued time displacement represents the current time when not otherwise changed by previous items. They may be used to stress other items, like in '`12:00 today`'. The string '`this`' also has the meaning of a zero-valued time displacement, but is preferred in date strings like '`this thursday`'.

When a relative item causes the resulting date to cross a boundary where the clocks were adjusted, typically for daylight-saving time, the resulting date and time are adjusted accordingly.

## 4.7 Pure numbers in date strings

The precise interpretation of a pure decimal number depends on the context in the date string.

If the decimal number is of the form *yyyymmdd* and no other calendar date item (see Section 4.2 [Calendar date items], page 13) appears before it in the date string, then *yyyy* is read as the year, *mm* as the month number and *dd* as the day of the month, for the specified calendar date.

If the decimal number is of the form *hhmm* and no other time of day item appears before it in the date string, then *hh* is read as the hour of the day and *mm* as the minute of the hour, for the specified time of the day. *mm* can also be omitted.

If both a calendar date and a time of day appear to the left of a number in the date string, but no relative item, then the number overrides the year.

## 4.8 Authors of `getdate`

`getdate` was originally implemented by Steven M. Bellovin (`smb@research.att.com`) while at the University of North Carolina at Chapel Hill. The code was later tweaked by a couple of people on Usenet, then completely overhauled by Rich $alz (`rsalz@bbn.com`) and Jim Berets (`jberets@bbn.com`) in August, 1990. Various revisions for the GNU system were made by David MacKenzie, Jim Meyering, Paul Eggert and others.

This chapter was originally produced by François Pinard (`pinard@iro.umontreal.ca`) from the '`getdate.y`' source code, and then edited by K. Berry (`kb@cs.umb.edu`).

# 5 Directory listing

This chapter describes the `ls` command and its variants `dir` and `vdir`, which list information about files.

## 5.1 `ls`: List directory contents

The `ls` program lists information about files (of any type, including directories). Options and file arguments can be intermixed arbitrarily, as usual.

For non-option command-line arguments that are directories, by default `ls` lists the contents of directories, not recursively, and omitting files with names beginning with '`.`'. For other non-option arguments, by default `ls` lists just the file name. If no non-option arguments are specified, `ls` lists the contents of the current directory.

By default, the output is sorted alphabetically. If standard output is a terminal, the output is in columns (sorted vertically) and control characters are output as question marks; otherwise, the output is listed one per line and control characters are output as-is.

Because `ls` is such a fundamental program, it has accumulated many options over the years. They are described in the subsections below; within each section, options are listed alphabetically (ignoring case). The division of options into the subsections is not absolute, since some options affect more than one aspect of `ls`'s operation.

The '`-g`' option is accepted but ignored, for compatibility with Unix. Also see Chapter 2 [Common options], page 2.

### 5.1.1 Which files are listed

These options determine which files `ls` lists information for. By default, any files and the contents of any directories on the command line are shown.

'`-a`'
'`--all`' List all files in directories, including files that start with '`.`'.

'`-A`'
'`--almost-all`'
List all files in directories except for '`.`' and '`..`'.

'`-B`'
'`--ignore-backups`'
Do not list files that end with '`~`', unless they are given on the command line.

'`-d`'
'`--directory`'
List just the names of directories, as with other types of files, rather than listing their contents.

'`-I PATTERN`'
'`--ignore=PATTERN`'
Do not list files whose names match the shell pattern (not regular expression) *pattern* unless they are given on the command line. As in the shell, an initial '`.`' in a file name does not match a wildcard at the start of *pattern*. Sometimes it is useful to give this option several times. For example,

```
$ ls --ignore='.??*' --ignore='.[^.]' --ignore='#*'
```

The first option ignores names of length 3 or more that start with '`.`', the second ignores all two-character names that start with '`.`' except '`..`', and the third ignores names that start with '`#`'.

'`-L`'
'`--dereference`'
: In a long listing, show file information (e.g., times and permissions) for the referents of symbolic links rather than for the symbolic links themselves.

'`-R`'
'`--recursive`'
: List the contents of all directories recursively.

### 5.1.2 What information is listed

These options affect the information that `ls` displays. By default, only file names are shown.

'`-D`'
'`--dired`'
: With the long listing ('`-l`') format, print an additional line after the main output:

`//DIRED//` *beg1 end1 beg2 end2 ...*

The *begN* and *endN* are unsigned integers that record the byte position of the beginning and end of each file name in the output. This makes it easy for Emacs to find the names, even when they contain unusual characters such as space or newline, without fancy searching.

If directories are being listed recursively (`-R`), output a similar line after each subdirectory:

`//SUBDIRED//` *format beg1 end1 ...*

Finally, output a line of the form:

`//DIRED-OPTIONS// --quoting-style=`*word*

where *word* is the quoting style (see Section 5.1.6 [Formatting the file names], page 23).

'`-G`'
'`--no-group`'
: Inhibit display of group information in a long format directory listing. (This is the default in some non-GNU versions of `ls`, so we provide this option for compatibility.)

'`-h`'
'`--human-readable`'
: Append a size letter such as '`M`' for megabytes to each size. Powers of 1024 are used, not 1000; '`M`' stands for 1,048,576 bytes. Use the '`--si`' option if you prefer powers of 1000.

'`-H`'
'`--si`'
: Append a size letter such as '`M`' for megabytes to each size. (SI is the International System of Units, which defines these letters as prefixes.) Powers

of 1000 are used, not 1024; 'M' stands for 1,000,000 bytes. Use the '-h' or '--human-readable' option if you prefer powers of 1024.

Warning: the meaning of '-H' will change in the future to conform to POSIX. Use '--si' for the old meaning.

'-i'
'--inode' Print the inode number (also called the file serial number and index number) of each file to the left of the file name. (This number uniquely identifies each file within a particular filesystem.)

'-l'
'--format=long'
'--format=verbose'
In addition to the name of each file, print the file type, permissions, number of hard links, owner name, group name, size in bytes, and timestamp (by default, the modification time). For files with a time more than six months old or in the future, the timestamp contains the year instead of the time of day. If the timestamp contains today's date with the year rather than a time of day, the file's time is in the future, which means you probably have clock skew problems which may break programs like `make` that rely on file times.

For each directory that is listed, preface the files with a line '`total` *blocks*', where *blocks* is the total disk allocation for all files in that directory. The block size currently defaults to 1024 bytes, but this can be overridden (see Section 2.2 [Block size], page 3). The *blocks* computed counts each hard link separately; this is arguably a deficiency.

The permissions listed are similar to symbolic mode specifications (see Section 3.2 [Symbolic Modes], page 6). But `ls` combines multiple bits into the third character of each set of permissions as follows:

's' If the setuid or setgid bit and the corresponding executable bit are both set.

'S' If the setuid or setgid bit is set but the corresponding executable bit is not set.

't' If the sticky bit and the other-executable bit are both set.

'T' If the sticky bit is set but the other-executable bit is not set.

'x' If the executable bit is set and none of the above apply.

'-' Otherwise.

Following the permission bits is a single character that specifies whether an alternate access method applies to the file. When that character is a space, there is no alternate access method. When it is a printing character (e.g., '+'), then there is such a method.

'-o' Produce long format directory listings, but don't display group information. It is equivalent to using '--format=long' with '--no-group' . This option is provided for compatibility with other versions of `ls`.

'-s'
'--size' Print the disk allocation of each file to the left of the file name. This is the amount of disk space used by the file, which is usually a bit more than the file's size, but it can be less if the file has holes.

Normally the disk allocation is printed in units of 1024 bytes, but this can be overridden (see Section 2.2 [Block size], page 3).

For files that are NFS-mounted from an HP-UX system to a BSD system, this option reports sizes that are half the correct values. On HP-UX systems, it reports sizes that are twice the correct values for files that are NFS-mounted from BSD systems. This is due to a flaw in HP-UX; it also affects the HP-UX `ls` program.

### 5.1.3 Sorting the output

These options change the order in which `ls` sorts the information it outputs. By default, sorting is done by character code (e.g., ASCII order).

'-c'
'--time=ctime'
'--time=status'
'--time=use'
If the long listing format (e.g., '-l', '-o') is being used, print the status change time (the 'ctime' in the inode) instead of the modification time. When explicitly sorting by time ('--sort=time' or '-t') or when not using a long listing format, sort according to the status change time.

'-f' Primarily, like '-U'—do not sort; list the files in whatever order they are stored in the directory. But also enable '-a' (list all files) and disable '-l', '--color', and '-s' (if they were specified before the '-f').

'-r'
'--reverse'
Reverse whatever the sorting method is—e.g., list files in reverse alphabetical order, youngest first, smallest first, or whatever.

'-S'
'--sort=size'
Sort by file size, largest first.

'-t'
'--sort=time'
Sort by modification time (the 'mtime' in the inode), newest first.

'-u'
'--time=atime'
'--time=access'
If the long listing format (e.g., '--format=long') is being used, print the last access time (the 'atime' in the inode). When explicitly sorting by time ('--sort=time' or '-t') or when not using a long listing format, sort according to the access time.

'-U'
'--sort=none'
Do not sort; list the files in whatever order they are stored in the directory. (Do not do any of the other unrelated things that '-f' does.) This is especially useful when listing very large directories, since not doing any sorting can be noticeably faster.

'-v'
'--sort=version'
Sort by version name and number, lowest first. It behaves like a default sort, except that each sequence of decimal digits is treated numerically as an index/version number. (See Section 5.1.4 [More details about version sort], page 21.)

'-X'
'--sort=extension'
Sort directory contents alphabetically by file extension (characters after the last '.'); files with no extension are sorted first.

### 5.1.4 More details about version sort

The version sort takes into account the fact that file names frequently include indices or version numbers. Standard sorting functions usually do not produce the ordering that people expect because comparisons are made on a character-by-character basis. The version sort addresses this problem, and is especially useful when browsing directories that contain many files with indices/version numbers in their names:

```
> ls -1              > ls -1v
foo.zml-1.gz         foo.zml-1.gz
foo.zml-100.gz       foo.zml-2.gz
foo.zml-12.gz        foo.zml-6.gz
foo.zml-13.gz        foo.zml-12.gz
foo.zml-2.gz         foo.zml-13.gz
foo.zml-25.gz        foo.zml-25.gz
foo.zml-6.gz         foo.zml-100.gz
```

Note also that numeric parts with leading zeroes are considered as fractional one:

```
> ls -1              > ls -1v
abc-1.007.tgz        abc-1.007.tgz
abc-1.012b.tgz       abc-1.01a.tgz
abc-1.01a.tgz        abc-1.012b.tgz
```

### 5.1.5 General output formatting

These options affect the appearance of the overall output.

'-1'
'--format=single-column'
List one file per line. This is the default for ls when standard output is not a terminal.

'-C'
'--format=vertical'
: List files in columns, sorted vertically. This is the default for `ls` if standard output is a terminal. It is always the default for the `dir` and `d` programs. GNU `ls` uses variable width columns to display as many files as possible in the fewest lines.

'--color [=*when*]'
: Specify whether to use color for distinguishing file types. *when* may be omitted, or one of:
  - none - Do not use color at all. This is the default.
  - auto - Only use color if standard output is a terminal.
  - always - Always use color.

  Specifying '--color' and no *when* is equivalent to '--color=always'. Piping a colorized listing through a pager like `more` or `less` usually produces unreadable results. However, using `more -f` does seem to work.

'-F'
'--classify'
'--indicator-style=classify'
: Append a character to each file name indicating the file type. Also, for regular files that are executable, append '*'. The file type indicators are '/' for directories, '@' for symbolic links, '|' for FIFOs, '=' for sockets, and nothing for regular files.

'--full-time'
: List times in full, rather than using the standard abbreviation heuristics. The format is currently similar to that of `date`, but this is planned to change in a future release, partly because modern file time stamps have more precision. It's not possible to change the format, but you can extract out the date string with `cut` and then pass the result to `date -d`. See section "date invocation" in *Shell utilities*.

  This is most useful because the time output includes the seconds. (Unix filesystems store file timestamps only to the nearest second, so this option shows all the information there is.) For example, this can help when you have a Makefile that is not regenerating files properly.

'--indicator-style=*word*'
: Append a character indicator with style *word* to entry names, as follows:

  'none'
  : Do not append any character indicator; this is the default.

  'file-type'
  : Append '/' for directories, '@' for symbolic links, '|' for FIFOs, '=' for sockets, and nothing for regular files. This is the same as the '-p' or '--file-type' option.

  'classify'
  : Append '*' for executable regular files, otherwise behave as for 'file-type'. This is the same as the '-F' or '--classify' option.

'-k'
'--kilobytes'
Print file sizes in 1024-byte blocks, overriding the default block size (see Section 2.2 [Block size], page 3).

'-m'
'--format=commas'
List files horizontally, with as many as will fit on each line, separated by ', ' (a comma and a space).

'-n'
'--numeric-uid-gid'
List the numeric UID and GID instead of the names.

'-p'
'--file-type'
'--indicator-style=file-type'
Append a character to each file name indicating the file type. This is like '-F', except that executables are not marked.

'-x *format*'
'--format=across'
'--format=horizontal'
List the files in columns, sorted horizontally.

'-T *cols*'
'--tabsize=*cols*'
Assume that each tabstop is *cols* columns wide. The default is 8. `ls` uses tabs where possible in the output, for efficiency. If *cols* is zero, do not use tabs at all.

'-w'
'--width=*cols*'
Assume the screen is *cols* columns wide. The default is taken from the terminal settings if possible; otherwise the environment variable `COLUMNS` is used if it is set; otherwise the default is 80.

### 5.1.6 Formatting the file names

These options change how file names themselves are printed.

'-b'
'--escape'
'--quoting-style=escape'
Quote nongraphic characters in file names using alphabetic and octal backslash sequences like those used in C.

'-N'
'--literal'
Do not quote file names.

'-q'
'--hide-control-chars'
Print question marks instead of nongraphic characters in file names. This is the default if the output is a terminal and the program is `ls`.

'-Q'
'--quote-name'
'--quoting-style=c'
Enclose file names in double quotes and quote nongraphic characters as in C.

'--quoting-style=*word*'
Use style *word* to quote output names. The *word* should be one of the following:

'literal' Output names as-is.

'shell' Quote names for the shell if they contain shell metacharacters or would cause ambiguous output.

'shell-always'
Quote names for the shell, even if they would normally not require quoting.

'c' Quote names as for a C language string; this is the same as the '-Q' or '--quote-name' option.

'escape' Quote as with 'c' except omit the surrounding double-quote characters; this is the same as the '-b' or '--escape' option.

'clocale' Quote as with 'c' except use quotation marks appropriate for the locale.

'locale' Like 'clocale', but quote `'like this'` instead of `"like this"` in the default C locale. This looks nicer on many displays.

You can specify the default value of the '--quoting-style' option with the environment variable `QUOTING_STYLE`. If that environment variable is not set, the default value is 'literal', but this default may change to 'shell' in a future version of this package.

'--show-control-chars'
Print nongraphic characters as-is in file names. This is the default unless the output is a terminal and the program is `ls`.

## 5.2 `dir`: Briefly list directory contents

`dir` (also installed as `d`) is equivalent to `ls -C -b`; that is, by default files are listed in columns, sorted vertically, and special characters are represented by backslash escape sequences.

See Section 5.1 [ls invocation], page 17.

## 5.3 `vdir`: Verbosely list directory contents

`vdir` (also installed as `v`) is equivalent to `ls -l -b`; that is, by default files are listed in long format and special characters are represented by backslash escape sequences.

## 5.4 `dircolors`: Color setup for `ls`

`dircolors` outputs a sequence of shell commands to set up the terminal for color output from `ls` (and `dir`, etc.). Typical usage:

`eval ‘dircolors [`*option*`]... [`*file*`]‘`

If *file* is specified, `dircolors` reads it to determine which colors to use for which file types and extensions. Otherwise, a precompiled database is used. For details on the format of these files, run ‘`dircolors --print-database`’.

The output is a shell command to set the `LS_COLORS` environment variable. You can specify the shell syntax to use on the command line, or `dircolors` will guess it from the value of the `SHELL` environment variable.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

‘`-b`’
‘`--sh`’
‘`--bourne-shell`’
: Output Bourne shell commands. This is the default if the `SHELL` environment variable is set and does not end with ‘`csh`’ or ‘`tcsh`’.

‘`-c`’
‘`--csh`’
‘`--c-shell`’
: Output C shell commands. This is the default if `SHELL` ends with `csh` or `tcsh`.

‘`-p`’
‘`--print-database`’
: Print the (compiled-in) default color configuration database. This output is itself a valid configuration file, and is fairly descriptive of the possibilities.

# 6 Basic operations

This chapter describes the commands for basic file manipulation: copying, moving (renaming), and deleting (removing).

## 6.1 `cp`: Copy files and directories

`cp` copies files (or, optionally, directories). The copy is completely independent of the original. You can either copy one file to another, or copy arbitrarily many files to a destination directory. Synopsis:

> `cp [`*option*`]`... *source dest*
> `cp [`*option*`]`... *source*... *directory*

If the last argument names an existing directory, `cp` copies each *source* file into that directory (retaining the same name). Otherwise, if only two files are given, it copies the first onto the second. It is an error if the last argument is not a directory and more than two non-option arguments are given.

Generally, files are written just as they are read. For exceptions, see the '`--sparse`' option below.

By default, `cp` does not copy directories. However, the '`-R`', '`-a`', and '`-r`' options cause `cp` to copy recursively by descending into source directories and copying files to corresponding destination directories.

By default, `cp` follows symbolic links only when not copying recursively. This default can be overridden with the '`--no-dereference`' ('`-d`'), '`--dereference`' ('`-L`'), and '`-H`' options. If more than one of these options is specified, the last one silently overrides the others.

`cp` generally refuses to copy a file onto itself, with the following exception: if '`--force --backup`' is specified with *source* and *dest* identical, and referring to a regular file, `cp` will make a backup file, either regular or numbered, as specified in the usual ways (see Section 2.1 [Backup options], page 2). This is useful when you simply want to make a backup of an existing file before changing it.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-a`'
'`--archive`'
: Preserve as much as possible of the structure and attributes of the original files in the copy (but do not attempt to preserve internal directory structure; i.e., '`ls -U`' may list the entries in a copied directory in a different order). Equivalent to '`-dpR`'.

'`-b`'
'`--backup[=`*method*`]`'
: See Section 2.1 [Backup options], page 2. Make a backup of each file that would otherwise be overwritten or removed. As a special case, `cp` makes a backup of *source* when the force and backup options are given and *source* and *dest* are the same name for an existing, regular file. One useful application of this combination of options is this tiny Bourne shell script:

```
#!/bin/sh
# Usage: backup FILE...
# Create a GNU-style backup of each listed FILE.
for i in "$"; do
  cp --backup --force "$i" "$i"
done
```

'-d'
'--no-dereference'
: Copy symbolic links as symbolic links rather than copying the files that they point to, and preserve hard links between source files in the copies.

'-f'
'--force'
: When copying without this option and an existing destination file cannot be opened for writing, the copy fails. However, with '--force'), when a destination file cannot be opened, `cp` then unlinks it and tries to open it again. Contrast this behavior with that enabled by '--link' and '--symbolic-link', whereby the destination file is never opened but rather is unlinked unconditionally. Also see the description of '--remove-destination'.

'-H'
: If a command line argument specifies a symbolic link, then copy the file it points to rather than the symbolic link itself. However, copy (preserving its nature) any symbolic link that is encountered via recursive traversal.

'-i'
'--interactive'
: Prompt whether to overwrite existing regular destination files.

'-l'
'--link'
: Make hard links instead of copies of non-directories.

'-L'
'--dereference'
: Always follow symbolic links.

'-p'
'--preserve'
: Preserve the original files' owner, group, permissions, and timestamps. In the absence of this option, each destination file is created with the permissions of the corresponding source file, minus the bits set in the umask. See Chapter 3 [File permissions], page 6.

'-P'
'--parents'
: Form the name of each destination file by appending to the target directory a slash and the specified name of the source file. The last argument given to `cp` must be the name of an existing directory. For example, the command:

```
cp --parents a/b/c existing_dir
```

copies the file 'a/b/c' to 'existing_dir/a/b/c', creating any missing intermediate directories.

Warning: the meaning of '-P' will change in the future to conform to POSIX. Use '--parents' for the old meaning, and '--no-dereference' for the new.

'-r' Copy directories recursively, copying any non-directories and special files (e.g., symbolic links, FIFOs and device files) as if they were regular files. This means trying to read the data in each source file and writing it to the destination. It is usually a mistake to apply `cp -r` to special files like FIFOs and the ones typically found in the '/dev' directory. In most cases, `cp -r` will hang indefinitely trying to read from FIFOs and special files like '/dev/console', and it will fill up your destination disk if you use it to copy '/dev/zero'. Use the '--recursive' ('-R') option instead if you want to copy special files, preserving their special nature rather than reading from them to copy their contents.

'-R'
'--recursive'
Copy directories recursively, preserving non-directories (contrast with '-r' just above).

'--remove-destination'
Remove each existing destination file before attempting to open it (contrast with '-f' above).

'--sparse=*when*'
A *sparse file* contains *holes*—a sequence of zero bytes that does not occupy any physical disk blocks; the 'read' system call reads these as zeroes. This can both save considerable disk space and increase speed, since many binary files contain lots of consecutive zero bytes. By default, `cp` detects holes in input source files via a crude heuristic and makes the corresponding output file sparse as well.

The *when* value can be one of the following:

'auto' The default behavior: the output file is sparse if the input file is sparse.

'always' Always make the output file sparse. This is useful when the input file resides on a filesystem that does not support sparse files (the most notable example is 'efs' filesystems in SGI IRIX 5.3 and earlier), but the output file is on another type of filesystem.

'never' Never make the output file sparse. This is useful in creating a file for use with the `mkswap` command, since such a file must not have any holes.

'--*strip-trailing-slashes*'
Remove any trailing slashes from each *source* argument. See Section 2.4 [Trailing slashes], page 5.

'-s'
'--symbolic-link'
Make symbolic links instead of copies of non-directories. All source file names must be absolute (starting with '/') unless the destination files are in the current directory. This option merely results in an error message on systems that do not support symbolic links.

'-S *suffix*'
'--suffix=*suffix*'
: Append *suffix* to each backup file made with '-b'. See Section 2.1 [Backup options], page 2.

'--target-directory=*directory*'
: Specify the destination *directory*. See Section 2.3 [Target directory], page 4.

'-v'
'--verbose'
: Print the name of each file before copying it.

'-V *method*'
'--version-control=*method*'
: Change the type of backups made with '-b'. The *method* argument can be 'none' (or 'off'), 'numbered' (or 't'), 'existing' (or 'nil'), or 'never' (or 'simple'). See Section 2.1 [Backup options], page 2.

'-x'
'--one-file-system'
: Skip subdirectories that are on different filesystems from the one that the copy started on. However, mount point directories *are* copied.

## 6.2 dd: Convert and copy a file

dd copies a file (from standard input to standard output, by default) with a changeable I/O block size, while optionally performing conversions on it. Synopsis:

```
dd [option]...
```

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

The numeric-valued options below (*bytes* and *blocks*) can be followed by a multiplier: 'b'=512, 'c'=1, 'w'=2, 'x*m*'=*m*, or any of the standard block size suffixes like 'k'=1024 (see Section 2.2 [Block size], page 3).

Use different dd invocations to use different block sizes for skipping and I/O. For example, the following shell commands copy data in 512 kB blocks between a disk and a tape, but do not save or restore a 4 kB label at the start of the disk:

```
disk=/dev/rdsk/c0t1d0s2
tape=/dev/rmt/0

# Copy all but the label from disk to tape.
(dd bs=4k skip=1 count=0 && dd bs=512k) <$disk >$tape

# Copy from tape back to disk, but leave the disk label alone.
(dd bs=4k seek=1 count=0 && dd bs=512k) <$tape >$disk
```

'if=*file*'
: Read from *file* instead of standard input.

'of=*file*'
: Write to *file* instead of standard output. Unless 'conv=notrunc' is given, dd truncates *file* to zero bytes (or the size specified with 'seek=').

'`ibs=`*bytes*'
: Read *bytes* bytes at a time.

'`obs=`*bytes*'
: Write *bytes* bytes at a time.

'`bs=`*bytes*' Both read and write *bytes* bytes at a time. This overrides '`ibs`' and '`obs`'.

'`cbs=`*bytes*'
: Convert *bytes* bytes at a time.

'`skip=`*blocks*'
: Skip *blocks* '`ibs`'-byte blocks in the input file before copying.

'`seek=`*blocks*'
: Skip *blocks* '`obs`'-byte blocks in the output file before copying.

'`count=`*blocks*'
: Copy *blocks* '`ibs`'-byte blocks from the input file, instead of everything until the end of the file.

'`conv=`*conversion*`[,`*conversion*`]...`'
: Convert the file as specified by the *conversion* argument(s). (No spaces around any comma(s).)

Conversions:

'`ascii`' Convert EBCDIC to ASCII.

'`ebcdic`' Convert ASCII to EBCDIC.

'`ibm`' Convert ASCII to alternate EBCDIC.

'`block`' For each line in the input, output '`cbs`' bytes, replacing the input newline with a space and padding with spaces as necessary.

'`unblock`' Replace trailing spaces in each '`cbs`'-sized input block with a newline.

'`lcase`' Change uppercase letters to lowercase.

'`ucase`' Change lowercase letters to uppercase.

'`swab`' Swap every pair of input bytes. GNU `dd`, unlike others, works when an odd number of bytes are read—the last byte is simply copied (since there is nothing to swap it with).

'`noerror`' Continue after read errors.

'`notrunc`' Do not truncate the output file.

'`sync`' Pad every input block to size of '`ibs`' with trailing zero bytes. When use with '`block`' or '`unblock`', pad with spaces instead of zero bytes.

## 6.3 `install`: Copy files and set attributes

`install` copies files while setting their permission modes and, if possible, their owner and group. Synopses:

`install` [*option*]... *source dest*
`install` [*option*]... *source*... *directory*
`install -d` [*option*]... *directory*...

In the first of these, the *source* file is copied to the *dest* target file. In the second, each of the *source* files are copied to the destination *directory*. In the last, each *directory* (and any missing parent directories) is created.

`install` is similar to `cp`, but allows you to control the attributes of destination files. It is typically used in Makefiles to copy programs into their destination directories. It refuses to copy files onto themselves.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b`'
'`--backup[=`*method*`]`'
: See Section 2.1 [Backup options], page 2. Make a backup of each file that would otherwise be overwritten or removed.

'`-c`'
: Ignored; for compatibility with old Unix versions of `install`.

'`-d`'
'`--directory`'
: Create each given directory and any missing parent directories, setting the owner, group and mode as given on the command line or to the defaults. It also gives any parent directories it creates those attributes. (This is different from the SunOS 4.x `install`, which gives directories that it creates the default attributes.)

'`-g` *group*'
'`--group=`*group*'
: Set the group ownership of installed files or directories to *group*. The default is the process' current group. *group* may be either a group name or a numeric group id.

'`-m` *mode*'
'`--mode=`*mode*'
: Set the permissions for the installed file or directory to *mode*, which can be either an octal number, or a symbolic mode as in `chmod`, with 0 as the point of departure (see Chapter 3 [File permissions], page 6). The default mode is '`u=rwx,go=rx`'—read, write, and execute for the owner, and read and execute for group and other.

'`-o` *owner*'
'`--owner=`*owner*'
: If `install` has appropriate privileges (is run as root), set the ownership of installed files or directories to *owner*. The default is `root`. *owner* may be either a user name or a numeric user ID.

'-p'
'--preserve-timestamps'
: Set the time of last access and the time of last modification of each installed file to match those of each corresponding original file. When a file is installed without this option, its last access and last modification times are both set to the time of installation. This option is useful if you want to use the last modification times of installed files to keep track of when they were last built as opposed to when they were last installed.

'-s'
'--strip' Strip the symbol tables from installed binary executables.

'-S *suffix*'
'--suffix=*suffix*'
: Append *suffix* to each backup file made with '-b'. See Section 2.1 [Backup options], page 2.

'--target-directory=*directory*'
: Specify the destination *directory*. See Section 2.3 [Target directory], page 4.

'-v'
'--verbose'
: Print the name of each file before copying it.

'-V *method*'
'--version-control=*method*'
: Change the type of backups made with '-b'. The *method* argument can be 'none' (or 'off'), 'numbered' (or 't'), 'existing' (or 'nil'), or 'never' (or 'simple'). See Section 2.1 [Backup options], page 2.

## 6.4 mv: Move (rename) files

mv moves or renames files (or directories). Synopsis:

```
mv [option]... source dest
mv [option]... source... directory
```

If the last argument names an existing directory, mv moves each other given file into a file with the same name in that directory. Otherwise, if only two files are given, it renames the first as the second. It is an error if the last argument is not a directory and more than two files are given.

mv can move any type of file from one filesystem to another. Prior to version 4.0 of the fileutils, mv could move only regular files between filesystems. For example, now mv can move an entire directory hierarchy including special device files from one partition to another. It first uses some of the same code that's used by cp -a to copy the requested directories and files, then (assuming the copy succeeded) it removes the originals. If the copy fails, then the part that was copied to the destination partition is removed. If you were to copy three directories from one partition to another and the copy of the first directory succeeded, but the second didn't, the first would be left on the destination partion and the second and third would be left on the original partition.

If a destination file exists but is normally unwritable, standard input is a terminal, and the '-f' or '--force' option is not given, mv prompts the user for whether to replace the file. (You might own the file, or have write permission on its directory.) If the response does not begin with '`y`' or '`Y`', the file is skipped.

*Warning*: If you try to move a symlink that points to a directory, and you specify the symlink with a trailing slash, then `mv` doesn't move the symlink but instead moves the directory referenced by the symlink. See Section 2.4 [Trailing slashes], page 5.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b`'
'`--backup[=`*method*`]`'
: See Section 2.1 [Backup options], page 2. Make a backup of each file that would otherwise be overwritten or removed.

'`-f`'
'`--force`'
: Do not prompt the user before removing an unwritable destination file.

'`-i`'
'`--interactive`'
: Prompt whether to overwrite each existing destination file, regardless of its permissions. If the response does not begin with '`y`' or '`Y`', the file is skipped.

'`-u`'
'`--update`'
: Do not move a nondirectory that has an existing destination with the same or newer modification time.

'`-v`'
'`--verbose`'
: Print the name of each file before moving it.

'`--strip-trailing-slashes`'
: Remove any trailing slashes from each *source* argument. See Section 2.4 [Trailing slashes], page 5.

'`-S` *suffix*'
'`--suffix=`*suffix*'
: Append *suffix* to each backup file made with '`-b`'. See Section 2.1 [Backup options], page 2.

'`--target-directory=`*directory*'
: Specify the destination *directory*. See Section 2.3 [Target directory], page 4.

'`-V` *method*'
'`--version-control=`*method*'
: Change the type of backups made with '`-b`'. The *method* argument can be '`none`' (or '`off`'), '`numbered`' (or '`t`'), '`existing`' (or '`nil`'), or '`never`' (or '`simple`'). See Section 2.1 [Backup options], page 2.

## 6.5 `rm`: Remove files or directories

`rm` removes each given *file*. By default, it does not remove directories. Synopsis:

```
rm [option]... [file]...
```

If a file is unwritable, standard input is a terminal, and the '-f' or '--force' option is not given, or the '-i' or '--interactive' option *is* given, rm prompts the user for whether to remove the file. If the response does not begin with 'y' or 'Y', the file is skipped.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'-d'
'--directory'
: Attempt to remove directories with unlink instead of rmdir, and don't require a directory to be empty before trying to unlink it. This works only if you have appropriate privileges and if your operating system supports unlink for directories. Because unlinking a directory causes any files in the deleted directory to become unreferenced, it is wise to fsck the filesystem after doing this.

'-f'
'--force'
: Ignore nonexistent files and never prompt the user. Ignore any previous '--interactive' ('-i') option.

'-i'
'--interactive'
: Prompt whether to remove each file. If the response does not begin with 'y' or 'Y', the file is skipped. Ignore any previous '--force' ('-f') option.

'-r'
'-R'
'--recursive'
: Remove the contents of directories recursively.

'-v'
'--verbose'
: Print the name of each file before removing it.

One common question is how to remove files whose names begin with a '-'. GNU rm, like every program that uses the getopt function to parse its arguments, lets you use the '--' option to indicate that all following arguments are non-options. To remove a file called '-f' in the current directory, you could type either:

```
rm -- -f
```

or:

```
rm ./-f
```

The Unix rm program's use of a single '-' for this purpose predates the development of the getopt standard syntax.

## 6.6 shred: Remove files more securely

shred overwrites devices or files, to help prevent even very expensive hardware from recovering the data.

Ordinarily when you remove a file (see Section 6.5 [rm invocation], page 33), the data is not actually destroyed. Only the index listing where the file is stored is destroyed, and the storage is made available for reuse. There are undelete utilities that will attempt to reconstruct the index and can bring the file back if the parts were not reused.

On a busy system with a nearly-full drive, space can get reused in a few seconds. But there is no way to know for sure. If you have sensitive data, you may want to be sure that recovery is not possible by actually overwriting the file with non-sensitive data.

However, even after doing that, it is possible to take the disk back to a laboratory and use a lot of sensitive (and expensive) equipment to look for the faint "echoes" of the original data underneath the overwritten data. If the data has only been overwritten once, it's not even that hard.

The best way to remove something irretrievably is to destroy the media it's on with acid, melt it down, or the like. For cheap removable media like floppy disks, this is the preferred method. However, hard drives are expensive and hard to melt, so the `shred` utility tries to achieve a similar effect non-destructively.

This uses many overwrite passes, with the data patterns chosen to maximize the damage they do to the old data. While this will work on floppies, the patterns are designed for best effect on hard drives. For more details, see the source code and Peter Gutmann's paper *Secure Deletion of Data from Magnetic and Solid-State Memory*, from the proceedings of the Sixth USENIX Security Symposium (San Jose, California, 22–25 July, 1996). The paper is also available online `http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`.

**Please note** that `shred` relies on a very important assumption: that the filesystem overwrites data in place. This is the traditional way to do things, but many modern filesystem designs do not satisfy this assumption. Exceptions include:

- Log-structured or journaled filesystems, such as those supplied with AIX and Solaris.
- Filesystems that write redundant data and carry on even if some writes fail, such as RAID-based filesystems.
- Filesystems that make snapshots, such as Network Appliance's NFS server.
- Filesystems that cache in temporary locations, such as NFS version 3 clients.
- Compressed filesystems.

If you are not sure how your filesystem operates, then you should assume that it does not overwrite data in place, which means that shred cannot reliably operate on regular files in your filesystem.

Generally speaking, it is more reliable to shred a device than a file, since this bypasses the problem of filesystem design mentioned above. However, even shredding devices is not always completely reliable. For example, most disks map out bad sectors invisibly to the application; if the bad sectors contain sensitive data, `shred` won't be able to destroy it.

`shred` makes no attempt to detect or report these problem, just as it makes no attempt to do anything about backups. However, since it is more reliable to shred devices than files, `shred` by default does not truncate or remove the output file. This default is more suitable for devices, which typically cannot be truncated and should not be removed.

`shred [`*option*`]... `*file*`[...]`

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'-f'
'--force' Override file permissions if necessary to allow overwriting.

'-*NUMBER*'
'-n *NUMBER*'
'--iterations=*NUMBER*'
By default, `shred` uses 25 passes of overwrite. This is enough for all of the useful overwrite patterns to be used at least once. You can reduce this to save time, or increase it if you have a lot of time to waste.

'-s *BYTES*'
'--size=*BYTES*'
Shred the first *BYTES* bytes of the file. The default is to shred the whole file. *BYTES* can be followed by a size specification like 'k', 'M', or 'G' to specify a multiple. See Section 2.2 [Block size], page 3.

'-u'
'--remove'
After shredding a file, truncate it (if possible) and then remove it. If a file has multiple links, only the named links will be removed.

'-v'
'--verbose'
Display status updates as sterilization proceeds.

'-x'
'--exact' Normally, shred rounds the file size up to the next multiple of the filesystem block size to fully erase the last block of the file. This option suppresses that behavior. Thus, by default if you shred a 10-byte file on a system with 512-byte blocks, the resulting file will be 512 bytes long. With this option, shred does not increase the size of the file.

'-z'
'--zero' Normally, the last pass that `shred` writes is made up of random data. If this would be conspicuous on your hard drive (for example, because it looks like encrypted data), or you just think it's tidier, the '--zero' option adds an additional overwrite pass with all zero bits. This is in addition to the number of passes specified by the '--iterations' option.

'-' Shred standard output.

This argument is considered an option. If the common '--' option has been used to indicate the end of options on the command line, then '-' will be interpreted as an ordinary file name.

The intended use of this is to shred a removed temporary file. For example

```
i=`tempfile -m 0600`
exec 3<>"$i"
rm -- "$i"
echo "Hello, world" >&3
shred - >&3
exec 3>-
```

> Note that the shell command '`shred - >file`' does not shred the contents of *file*, since it truncates *file* before invoking `shred`. Use the command '`shred file`' or (if using a Bourne-compatible shell) the command '`shred - 1<>file`' instead.

You might use the following command to erase all trace of the file system you'd created on the floppy disk in your first drive. That command takes about 20 minutes to erase a 1.44MB floppy.

```
shred --verbose /dev/fd0
```

Similarly, to erase all data on a selected partition of your hard disk, you could give a command like this:

```
shred --verbose /dev/sda5
```

# 7 Special file types

This chapter describes commands which create special types of files (and `rmdir`, which removes directories, one special file type).

Although Unix-like operating systems have markedly fewer special file types than others, not *everything* can be treated only as the undifferentiated byte stream of *normal files*. For example, when a file is created or removed, the system must record this information, which it does in a *directory*—a special type of file. Although you can read directories as normal files, if you're curious, in order for the system to do its job it must impose a structure, a certain order, on the bytes of the file. Thus it is a "special" type of file.

Besides directories, other special file types include named pipes (FIFOs), symbolic links, sockets, and so-called *special files*.

## 7.1 `ln`: Make links between files

`ln` makes links between files. By default, it makes hard links; with the '`-s`' option, it makes symbolic (or *soft*) links. Synopses:

`ln [`*option*`]`... *target* `[`*linkname*`]`
`ln [`*option*`]`... *target*... *directory*

- If the last argument names an existing directory, `ln` creates a link to each *target* file in that directory, using the *targets*' names. (But see the description of the '`--no-dereference`' option below.)
- If two filenames are given, `ln` creates a link from the second to the first.
- If one *target* is given, `ln` creates a link to that file in the current directory.
- It is an error if the last argument is not a directory and more than two files are given. Without '`-f`' or '`-i`' (see below), `ln` will not remove an existing file. Use the '`--backup`' option to make `ln` rename existing files.

A *hard link* is another name for an existing file; the link and the original are indistinguishable. Technically speaking, they share the same inode, and the inode contains all the information about a file—indeed, it is not incorrect to say that the inode *is* the file. On all existing implementations, you cannot make a hard link to a directory, and hard links cannot cross filesystem boundaries. (These restrictions are not mandated by POSIX, however.)

*Symbolic links* (*symlinks* for short), on the other hand, are a special file type (which not all kernels support: System V release 3 (and older) systems lack symlinks) in which the link file actually refers to a different file, by name. When most operations (opening, reading, writing, and so on) are passed the symbolic link file, the kernel automatically *dereferences* the link and operates on the target of the link. But some operations (e.g., removing) work on the link file itself, rather than on its target. See section "Symbolic Links" in *The GNU C Library Reference Manual*.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-b`'
'`--backup[=`*method*`]`'
: See Section 2.1 [Backup options], page 2. Make a backup of each file that would otherwise be overwritten or removed.

'-d'
'-F'
'--directory'
: Allow the super-user to make hard links to directories.

'-f'
'--force'
: Remove existing destination files.

'-i'
'--interactive'
: Prompt whether to remove existing destination files.

'-n'
'--no-dereference'
: When given an explicit destination that is a symlink to a directory, treat that destination as if it were a normal file.

  When the destination is an actual directory (not a symlink to one), there is no ambiguity. The link is created in that directory. But when the specified destination is a symlink to a directory, there are two ways to treat the user's request. `ln` can treat the destination just as it would a normal directory and create the link in it. On the other hand, the destination can be viewed as a non-directory—as the symlink itself. In that case, `ln` must delete or backup that symlink before creating the new link. The default is to treat a destination that is a symlink to a directory just like a directory.

'-s'
'--symbolic'
: Make symbolic links instead of hard links. This option merely produces an error message on systems that do not support symbolic links.

'-S *suffix*'
'--suffix=*suffix*'
: Append *suffix* to each backup file made with '-b'. See Section 2.1 [Backup options], page 2.

'--target-directory=*directory*'
: Specify the destination *directory*. See Section 2.3 [Target directory], page 4.

'-v'
'--verbose'
: Print the name of each file before linking it.

'-V *method*'
'--version-control=*method*'
: Change the type of backups made with '-b'. The *method* argument can be 'none' (or 'off'), 'numbered' (or 't'), 'existing' (or 'nil'), or 'never' (or 'simple'). See Section 2.1 [Backup options], page 2.

Examples:

```
ln -s /some/name  # creates link ./name pointing to /some/name
ln -s /some/name myname  # creates link ./myname pointing to /some/name
ln -s a b ..      # creates links ../a and ../b pointing to ./a and ./b
```

## 7.2 `mkdir`: Make directories

`mkdir` creates directories with the specified names. Synopsis:

`mkdir [`*option*`]... `*name*...

If a *name* is an existing file but not a directory, `mkdir` prints a warning message on stderr and will exit with a status of 1 after processing any remaining *names*. The same is done when a *name* is an existing directory and the -p option is not given. If a *name* is an existing directory and the -p option is given, `mkdir` will ignore it. That is, `mkdir` will not print a warning, raise an error, or change the mode of the directory (even if the -m option is given), and will move on to processing any remaining *names*.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-m` *mode*'
'`--mode=`*mode*'
: Set the mode of created directories to *mode*, which is symbolic as in `chmod` and uses '`a=rwx`' (read, write and execute allowed for everyone) minus the bits set in the umask for the point of the departure. See Chapter 3 [File permissions], page 6.

'`-p`'
'`--parents`'
: Make any missing parent directories for each argument. The mode for parent directories is set to the umask modified by '`u+wx`'. Ignore arguments corresponding to existing directories.

'`-v`'

'`--verbose`'
: Print a message for each created directory. This is most useful with '`--parents`'.

## 7.3 `mkfifo`: Make FIFOs (named pipes)

`mkfifo` creates FIFOs (also called *named pipes*) with the specified names. Synopsis:

`mkfifo [`*option*`] `*name*...

A *FIFO* is a special file type that permits independent processes to communicate. One process opens the FIFO file for writing, and another for reading, after which data can flow as with the usual anonymous pipe in shells or elsewhere.

The program accepts the following option. Also see Chapter 2 [Common options], page 2.

'`-m` *mode*'
'`--mode=`*mode*'
: Set the mode of created FIFOs to *mode*, which is symbolic as in `chmod` and uses '`a=rw`' (read and write allowed for everyone) minus the bits set in the umask for the point of departure. See Chapter 3 [File permissions], page 6.

## 7.4 `mknod`: Make block or character special files

`mknod` creates a FIFO, character special file, or block special file with the specified name. Synopsis:

`mknod [`*option*`]`... *name type* `[`*major minor*`]`

Unlike the phrase "special file type" above, the term *special file* has a technical meaning on Unix: something that can generate or receive data. Usually this corresponds to a physical piece of hardware, e.g., a printer or a disk. (These files are typically created at system-configuration time.) The `mknod` command is what creates files of this type. Such devices can be read either a character at a time or a "block" (many characters) at a time, hence we say there are *block special* files and *character special* files.

The arguments after *name* specify the type of file to make:

'`p`' for a FIFO

'`b`' for a block special file

'`c`' for a character special file

When making a block or character special file, the major and minor device numbers must be given after the file type.

The program accepts the following option. Also see Chapter 2 [Common options], page 2.

'`-m` *mode*'
'`--mode=`*mode*'
: Set the mode of created files to *mode*, which is symbolic as in `chmod` and uses '`a=rw`' minus the bits set in the umask as the point of departure. See Chapter 3 [File permissions], page 6.

## 7.5 `rmdir`: Remove empty directories

`rmdir` removes empty directories. Synopsis:

`rmdir [`*option*`]`... *directory*...

If any *directory* argument does not refer to an existing empty directory, it is an error.

The program accepts the following option. Also see Chapter 2 [Common options], page 2.

'`--ignore-fail-on-non-empty`'
: Ignore each failure to remove a directory that is solely because the directory is non-empty.

'`-p`'
'`--parents`'
: Remove *directory*, then try to remove each component of *directory*. So, for example, '`rmdir -p a/b/c`' is similar to '`rmdir a/b/c a/b a`'. As such, it fails if any of those directories turns out not to be empty. Use the '`--ignore-fail-on-non-empty`' option to make it so such a failure does not evoke a diagnostic and does not cause `rmdir` to exit unsuccessfully.

'`-v`'

'--verbose'

Give a diagnostic for each successful removal. *directory* is removed.

See Section 6.5 [rm invocation], page 33, for how to remove non-empty directories (recursively).

# 8 Changing file attributes

A file is not merely its contents, a name, and a file type (see Chapter 7 [Special file types], page 38). A file also has an owner (a userid), a group (a group id), permissions (what the owner can do with the file, what people in the group can do, and what everyone else can do), various timestamps, and other information. Collectively, we call these a file's *attributes*.

These commands change file attributes.

## 8.1 `chown`: Change file owner and group

`chown` changes the user and/or group ownership of each given *file* to *new-owner* or to the user and group of an existing reference file. Synopsis:

`chown [`*option*`]... {`*new-owner* `| --reference=`*ref_file*`}` *file*...

If used, *new-owner* specifies the new owner and/or group as follows (with no embedded white space):

`[`*owner*`] [ [:] [`*group*`] ]`

Specifically:

*owner* If only an *owner* (a user name or numeric user id) is given, that user is made the owner of each given file, and the files' group is not changed.

*owner*'`:`'*group*
: If the *owner* is followed by a colon and a *group* (a group name or numeric group id), with no spaces between them, the group ownership of the files is changed as well (to *group*).

*owner*'`:`' If a colon but no group name follows *owner*, that user is made the owner of the files and the group of the files is changed to *owner*'s login group.

'`:`'*group* If the colon and following *group* are given, but the owner is omitted, only the group of the files is changed; in this case, `chown` performs the same function as `chgrp`.

You may use '`.`' in place of the '`:`' separator. This is a GNU extension for compatibility with older scripts. New scripts should avoid the use of '`.`' because GNU `chown` may fail if *owner* contains '`.`' characters.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-c`'
'`--changes`'
: Verbosely describe the action for each *file* whose ownership actually changes.

'`-f`'
'`--silent`'
'`--quiet`' Do not print error messages about files whose ownership cannot be changed.

'`--from=`*old-owner*'
: Change a *file*'s ownership only if it has current attributes specified by *old-owner*. *old-owner* has the same form as *new-owner* described above. This option is

useful primarily from a security standpoint in that it narrows considerably the window of potential abuse. For example, to reflect a UID numbering change for one user's files without an option like this, `root` might run

```
find / -owner OLDUSER -print0 | xargs -0 chown NEWUSER
```

But that is dangerous because the interval between when the `find` tests the existing file's owner and when the `chown` is actually run may be quite large. One way to narrow the gap would be to invoke chown for each file as it is found:

```
find / -owner OLDUSER -exec chown NEWUSER {} \;
```

But that is very slow if there are many affected files. With this option, it is safer (the gap is narrower still) though still not perfect:

```
chown -R --from=OLDUSER NEWUSER /
```

'`--dereference`'
: Do not act on symbolic links themselves but rather on what they point to.

'`-h`'
'`--no-dereference`'
: Act on symbolic links themselves instead of what they point to. This is the default. This mode relies on the `lchown` system call. On systems that do not provide the `lchown` system call, `chown` fails when a file specified on the command line is a symbolic link. By default, no diagnostic is issued for symbolic links encountered during a recursive traversal, but see '`--verbose`'.

'`--reference=`*ref_file*'
: Change the user and group of each *file* to be the same as those of *ref_file*. If *ref_file* is a symbolic link, do not use the user and group of the symbolic link, but rather those of the file it refers to.

'`-v`'
'`--verbose`'
: Output a diagnostic for every file processed. If a symbolic link is encountered during a recursive traversal on a system without the `lchown` system call, and '`--no-dereference`' is in effect, then issue a diagnostic saying neither the symbolic link nor its referent is being changed.

'`-R`'
'`--recursive`'
: Recursively change ownership of directories and their contents.

## 8.2 `chgrp`: Change group ownership

`chgrp` changes the group ownership of each given *file* to *group* (which can be either a group name or a numeric group id) or to the group of an existing reference file. Synopsis:

`chgrp [`*option*`]... {`*group* `| --reference=`*ref_file*`}` *file*...

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-c`'
'`--changes`'
: Verbosely describe the action for each *file* whose group actually changes.

'-f'
'--silent'
'--quiet' Do not print error messages about files whose group cannot be changed.

'--dereference'
Do not act on symbolic links themselves but rather on what they point to.

'-h'
'--no-dereference'
Act on symbolic links themselves instead of what they point to. This is the default. This mode relies on the `lchown` system call. On systems that do not provide the `lchown` system call, `chgrp` fails when a file specified on the command line is a symbolic link. By default, no diagnostic is issued for symbolic links encountered during a recursive traversal, but see '--verbose'.

'--reference=*ref_file*'
Change the group of each *file* to be the same as that of *ref_file*. If *ref_file* is a symbolic link, do not use the group of the symbolic link, but rather that of the file it refers to.

'-v'
'--verbose'
Output a diagnostic for every file processed. If a symbolic link is encountered during a recursive traversal on a system without the `lchown` system call, and '--no-dereference' is in effect, then issue a diagnostic saying neither the symbolic link nor its referent is being changed.

'-R'
'--recursive'
Recursively change the group ownership of directories and their contents.

## 8.3 chmod: Change access permissions

`chmod` changes the access permissions of the named files. Synopsis:

`chmod [`*option*`]... {`*mode*` | --reference=`*ref_file*`}` *file*...

`chmod` never changes the permissions of symbolic links, since the `chmod` system call cannot change their permissions. This is not a problem since the permissions of symbolic links are never used. However, for each symbolic link listed on the command line, `chmod` changes the permissions of the pointed-to file. In contrast, `chmod` ignores symbolic links encountered during recursive directory traversals.

If used, *mode* specifies the new permissions. For details, see the section on Chapter 3 [File permissions], page 6.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'-c'
'--changes'
Verbosely describe the action for each *file* whose permissions actually changes.

'-f'
'--silent'
'--quiet' Do not print error messages about files whose permissions cannot be changed.

'-v'
'--verbose'
Verbosely describe the action or non-action taken for every *file*.

'--reference=*ref_file*'
Change the mode of each *file* to be the same as that of *ref_file*. See Chapter 3 [File permissions], page 6. If *ref_file* is a symbolic link, do not use the mode of the symbolic link, but rather that of the file it refers to.

'-R'
'--recursive'
Recursively change permissions of directories and their contents.

## 8.4 touch: Change file timestamps

`touch` changes the access and/or modification times of the specified files. Synopsis:

`touch [`*option*`]... `*file*`...`

If the first *file* would be a valid argument to the '-t' option and no timestamp is given with any of the '-d', '-r', or '-t' options and the '--' argument is not given, that argument is interpreted as the time for the other files instead of as a file name. Warning: this usage is obsolescent, and future versions of POSIX will require that support for it be withdrawn. Use '-t' instead.

Any *file* that does not exist is created empty.

If changing both the access and modification times to the current time, `touch` can change the timestamps for files that the user running it does not own but has write permission for. Otherwise, the user must own the files.

Although `touch` provides options for changing two of the times – the times of last access and modification – of a file, there is actually a third one as well: the inode change time. This is often referred to as a file's `ctime`. The inode change time represents the time when the file's meta-information last changed. One common example of this is when the permissions of a file change. Changing the permissions doesn't access the file, so the atime doesn't change, nor does it modify the file, so the mtime doesn't change. Yet, something about the file itself has changed, and this must be noted somewhere. This is the job of the ctime field. This is necessary, so that, for example, a backup program can make a fresh copy of the file, including the new permissions value. Another operation that modifies a file's ctime without affecting the others is renaming. In any case, it is not possible, in normal operations, for a user to change the ctime field to a user-specified value.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'-a'
'--time=atime'
'--time=access'
'--time=use'
Change the access time only.

'-c'
'--no-create'
Do not create files that do not exist.

'-d'
'--date=time'
Use *time* instead of the current time. It can contain month names, timezones, 'am' and 'pm', etc. See Chapter 4 [Date input formats], page 12.

'-f' Ignored; for compatibility with BSD versions of `touch`.

'-m'
'--time=mtime'
'--time=modify'
Change the modification time only.

'-r *file*'
'--reference=*file*'
Use the times of the reference *file* instead of the current time.

'-t [[CC]YY]MMDDhhmm[.ss]'
Use the argument (optional four-digit or two-digit years, months, days, hours, minutes, optional seconds) instead of the current time. If the year is specified with only two digits, then *CC* is 20 for years in the range 0 ... 68, and 19 for years in 69 ... 99. If no digits of the year are specified, the argument is interpreted as a date in the current year.

# 9 Disk usage

No disk can hold an infinite amount of data. These commands report on how much disk storage is in use or available. (This has nothing much to do with how much *main memory*, i.e., RAM, a program is using when it runs; for that, you want `ps` or `pstat` or `swap` or some such command.)

## 9.1 `df`: Report filesystem disk space usage

`df` reports the amount of disk space used and available on filesystems. Synopsis:

`df [`*option*`]... [`*file*`]...`

With no arguments, `df` reports the space used and available on all currently mounted filesystems (of all types). Otherwise, `df` reports on the filesystem containing each argument *file*.

Normally the disk space is printed in units of 1024 bytes, but this can be overridden (see Section 2.2 [Block size], page 3).

If an argument *file* is a disk device file containing a mounted filesystem, `df` shows the space available on that filesystem rather than on the filesystem containing the device node (i.e., the root filesystem). GNU `df` does not attempt to determine the disk usage on unmounted filesystems, because on most kinds of systems doing so requires extremely non-portable intimate knowledge of filesystem structures.

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'`-a`'
'`--all`' Include in the listing filesystems that have a size of 0 blocks, which are omitted by default. Such filesystems are typically special-purpose pseudo-filesystems, such as automounter entries. Also, filesystems of type "ignore" or "auto", supported by some operating systems, are only included if this option is specified.

'`-h`'
'`--human-readable`'
Append a size letter such as '`M`' for megabytes to each size. Powers of 1024 are used, not 1000; '`M`' stands for 1,048,576 bytes. Use the '`-H`' or '`--si`' option if you prefer powers of 1000.

'`-H`'
'`--si`' Append a size letter such as '`M`' for megabytes to each size. (SI is the International System of Units, which defines these letters as prefixes.) Powers of 1000 are used, not 1024; '`M`' stands for 1,000,000 bytes. Use the '`-h`' or '`--human-readable`' option if you prefer powers of 1024.

'`-i`'
'`--inodes`'
List inode usage information instead of block usage. An inode (short for index node) contains information about a file such as its owner, permissions, timestamps, and location on the disk.

'-k'
'--kilobytes'
Print sizes in 1024-byte blocks, overriding the default block size (see Section 2.2 [Block size], page 3).

'-l'
'--local' Limit the listing to local filesystems. By default, remote filesystems are also listed.

'-m'
'--megabytes'
Print sizes in megabyte (that is, 1,048,576-byte) blocks.

'--no-sync'
Do not invoke the `sync` system call before getting any usage data. This may make `df` run significantly faster on systems with many disks, but on some systems (notably SunOS) the results may be slightly out of date. This is the default.

'-P'
'--portability'
Use the POSIX output format. This is like the default format except for the following:

1. The information about each filesystem is always printed on exactly one line; a mount device is never put on a line by itself. This means that if the mount device name is more than 20 characters long (e.g., for some network mounts), the columns are misaligned.
2. Non-integer values are rounded up, instead of being rounded down or rounded to the nearest integer.
3. The labels in the header output line are changed to conform to POSIX.

'--sync' Invoke the `sync` system call before getting any usage data. On some systems (notably SunOS), doing this yields more up to date results, but in general this option makes `df` much slower, especially when there are many or very busy filesystems.

'-t *fstype*'
'--type=*fstype*'
Limit the listing to filesystems of type *fstype*. Multiple filesystem types can be specified by giving multiple '-t' options. By default, nothing is omitted.

'-T'
'--print-type'
Print each filesystem's type. The types printed here are the same ones you can include or exclude with '-t' and '-x'. The particular types printed are whatever is supported by the system. Here are some of the common names (this list is certainly not exhaustive):

'nfs' An NFS filesystem, i.e., one mounted over a network from another machine. This is the one type name which seems to be used uniformly by all systems.

'4.2, ufs, efs...'
: A filesystem on a locally-mounted hard disk. (The system might even support more than one type here; Linux does.)

'hsfs, cdfs'
: A filesystem on a CD-ROM drive. HP-UX uses 'cdfs', most other systems use 'hsfs' ('hs' for "High Sierra").

'pcfs' An MS-DOS filesystem, usually on a diskette.

'-x *fstype*'
'--exclude-type=*fstype*'
: Limit the listing to filesystems not of type *fstype*. Multiple filesystem types can be eliminated by giving multiple '-x' options. By default, no filesystem types are omitted.

'-v' Ignored; for compatibility with System V versions of df.

## 9.2 du: Estimate file space usage

du reports the amount of disk space used by the specified files and for each subdirectory (of directory arguments). Synopsis:

du [*option*]... [*file*]...

With no arguments, du reports the disk space for the current directory. Normally the disk space is printed in units of 1024 bytes, but this can be overridden (see Section 2.2 [Block size], page 3).

The program accepts the following options. Also see Chapter 2 [Common options], page 2.

'-a'
'--all' Show counts for all files, not just directories.

'-b'
'--bytes' Print sizes in bytes, overriding the default block size (see Section 2.2 [Block size], page 3).

'-c'
'--total' Print a grand total of all arguments after all arguments have been processed. This can be used to find out the total disk usage of a given set of files or directories.

'-D'
'--dereference-args'
: Dereference symbolic links that are command line arguments. Does not affect other symbolic links. This is helpful for finding out the disk usage of directories, such as '/usr/tmp', which are often symbolic links.

'-h'
'--human-readable'
: Append a size letter such as 'M' for megabytes to each size. Powers of 1024 are used, not 1000; 'M' stands for 1,048,576 bytes. Use the '-H' or '--si' option if you prefer powers of 1000.

'-H'
'--si' Append a size letter such as 'M' for megabytes to each size. (SI is the International System of Units, which defines these letters as prefixes.) Powers of 1000 are used, not 1024; 'M' stands for 1,000,000 bytes. Use the '-h' or '--human-readable' option if you prefer powers of 1024.

'-k'
'--kilobytes'
Print sizes in 1024-byte blocks, overriding the default block size (see Section 2.2 [Block size], page 3).

'-l'
'--count-links'
Count the size of all files, even if they have appeared already (as a hard link).

'-L'
'--dereference'
Dereference symbolic links (show the disk space used by the file or directory that the link points to instead of the space used by the link).

'--max-depth=*DEPTH*'
Show the total for each directory (and file if –all) that is at most MAX_DEPTH levels down from the root of the hierarchy. The root is at level 0, so `du --max-depth=0` is equivalent to `du -s`.

'-m'
'--megabytes'
Print sizes in megabyte (that is, 1,048,576-byte) blocks.

'-s'
'--summarize'
Display only a total for each argument.

'-S'
'--separate-dirs'
Report the size of each directory separately, not including the sizes of subdirectories.

'-x'
'--one-file-system'
Skip directories that are on different filesystems from the one that the argument being processed is on.

'--exclude=*PAT*'
When recursing, skip subdirectories or files matching *PAT*. For example, `du --exclude='*.o'` excludes files whose names end in '.o'.

'-X *FILE*'
'--exclude-from=*FILE*'
Like '--exclude', except take the patterns to exclude from *FILE*, one per line. If *FILE* is '-', take the patterns from standard input.

On BSD systems, `du` reports sizes that are half the correct values for files that are NFS-mounted from HP-UX systems. On HP-UX systems, it reports sizes that are twice the correct values for files that are NFS-mounted from BSD systems. This is due to a flaw in HP-UX; it also affects the HP-UX `du` program.

## 9.3 `sync`: Synchronize data on disk with memory

`sync` writes any data buffered in memory out to disk. This can include (but is not limited to) modified superblocks, modified inodes, and delayed reads and writes. This must be implemented by the kernel; The `sync` program does nothing but exercise the `sync` system call.

The kernel keeps data in memory to avoid doing (relatively slow) disk reads and writes. This improves performance, but if the computer crashes, data may be lost or the filesystem corrupted as a result. `sync` ensures everything in memory is written to disk.

Any arguments are ignored, except for a lone '`--help`' or '`--version`' (see Chapter 2 [Common options], page 2).

# Index

## -

## D



## E



## F

## O



## P



## Q



## R



## S

## T



## U



## V



## W



## Y

# Table of Contents