-- --

IEEE P1003.2 Draft 11.2 – September 1991

4.13 `cp` — Copy files

4.13.1 Synopsis

`cp [−fip]` *source_file target_file* 2

`cp [−fip]` *source_file ... target* 2

`cp −R [−fip]` *source_file ... target* 2

`cp −r [−fip]` *source_file ... target* 2

4.13.2 Description

The first synopsis form is denoted by two operands, neither of which are existing files of type directory. The `cp` utility shall copy the contents of *source_file* to the destination path named by *target_file*.

The second synopsis form is denoted by two or more operands where the `−R` or `−r` options are not specified and the first synopsis form is not applicable. It shall be an error if any *source_file* is a file of type directory, if *target* does not exist, or if *target* is a file of a type defined by POSIX.1 {8}, but is not a file of type directory. The `cp` utility shall copy the contents of each *source_file* to the destination path named by the concatenation of *target*, a slash character, and the last component of *source_file*.

The third and fourth synopsis forms are denoted by two or more operands where the `−R` or `−r` options are specified. The `cp` utility shall copy each file in the file hierarchy rooted in each *source_file* to a destination path named as follows.

If *target* exists and is a file of type directory, the name of the corresponding destination path for each file in the file hierarchy shall be the concatenation of *target*, a slash character, and the pathname of the file relative to the directory containing *source_file*.

If *target* does not exist, and two operands are specified, the name of the corresponding destination path for *source_file* shall be *target*; the name of the corresponding destination path for all other files in the file hierarchy shall be the concatenation of *target*, a slash character, and the pathname of the file relative to *source_file*.

It shall be an error if *target* does not exist and more than two operands are specified, or if *target* exists and is a file of a type defined by POSIX.1 {8}, but is not a file of type directory.

In the following description, *source_file* refers to the file that is being copied, whether specified as an operand or a file in a file hierarchy rooted in a *source_file* operand. The term *dest_file* refers to the file named by the destination path.

For each *source_file*, the following steps shall be taken:

(1) If *source_file* references the same file as *dest_file,* `cp` may write a diagnostic message to standard error; it shall do nothing more with *source_file* 1 and shall go on to any remaining files. 1

(2) If *source_file* is of type directory, the following steps shall be taken:

(a) If neither the `−R` or `−r` options were specified, `cp` shall write a diagnostic message to standard error, do nothing more with *source_file,* and go on to any remaining files.

(b) If *source_file* was not specified as an operand and *source_file* is dot or dot-dot, `cp` shall do nothing more with *source_file* and go on to any remaining files.

(c) If *dest_file* exists and it is a file type not specified by POSIX.1 {8}, the behavior is implementation defined.

(d) If *dest_file* exists and it is not of type directory, `cp` shall write a diagnostic message to standard error, do nothing more with *source_file* or any files below *source_file* in the file hierarchy, and go on to any remaining files.

(e) If the directory *dest_file* does not exist, it shall be created with file permission bits set to the same value as those of *source_file*, modified by the file creation mask of the user if the `−p` option was not specified, and then bitwise inclusively ORed with S_IRWXU. If *dest_file* cannot be created, `cp` shall write a diagnostic message to standard error, do nothing more with *source_file,* and go on to any remaining files. It is unspecified if `cp` shall attempt to copy files in the file hierarchy rooted in *source_file.*

(f) The files in the directory *source_file* shall be copied to the directory *dest_file*, taking the four steps [(1)-(4)] listed here with the files as *source_file* s.

(g) If *dest_file* was created, its file permission bits shall be changed (if necessary) to be the same as those of *source_file,* modified by the file creation mask of the user if the `−p` option was not specified.

(h) The `cp` utility shall do nothing more with *source_file* and go on to any remaining files.

(3) If *source_file* is of type regular file, the following steps shall be taken: 1

(a) If *dest_file* exists, the following steps are taken:

[1] If the `−i` option is in effect, the `cp` utility shall write a prompt to the standard error and read a line from the standard input. If the response is not affirmative, `cp` shall do nothing more with *source_file* and go on to any remaining files.

[2] A file descriptor for *dest_file* shall be obtained by performing actions equivalent to the POSIX.1 {8} *open*() function call using *dest_file* as the *path* argument, and the bitwise inclusive OR of O_WRONLY and O_TRUNC as the *oflag* argument.

[3] If the attempt to obtain a file descriptor fails and the `-f` option is in effect, `cp` shall attempt to remove the file by performing actions equivalent to the POSIX.1 {8} *unlink*() function called using *dest_file* as the *path* argument. If this attempt succeeds, `cp` shall continue with step (3b).

(b) If *dest_file* does not exist, a file descriptor shall be obtained by performing actions equivalent to the POSIX.1 {8} *open*() function called using *dest_file* as the *path* argument, and the bitwise inclusive OR of O_WRONLY and O_CREAT as the *oflag* argument. The file permission bits of *source_file* shall be the *mode* argument.

(c) If the attempt to obtain a file descriptor fails, `cp` shall write a diagnostic message to standard error, do nothing more with *source_file,* and go on to any remaining files.

(d) The contents of *source_file* shall be written to the file descriptor. Any write errors shall cause `cp` to write a diagnostic message to standard error and continue to step (3)(e).

(e) The file descriptor shall be closed.

(f) The `cp` utility shall do nothing more with *source_file.* If a write error occurred in step (3d), it is unspecified if `cp` continues with any remaining files. If no write error occurred in step (3d), `cp` shall go on to any remaining files.

(4) Otherwise, the following steps shall be taken:

(a) If the `-r` option was specified, the behavior is implementation defined.

(b) If the `-R` option was specified, the following steps shall be taken:

[1] The *dest_file* shall be created with the same file type as *source_file* .

[2] If *source_file* is a file of type FIFO , the file permission bits shall be the same as those of *source_file,* modified by the file creation mask of the user if the `-p` option was not specified. Otherwise, the permissions, owner ID, and group ID of *dest_file* are implementation defined.

If this creation fails for any reason, `cp` shall write a diagnostic message to standard error, do nothing more with *source_file,* and go on to any remaining files.

If the implementation provides additional or alternate access control mechanisms (see 2.2.2.55), their effect on copies of files is implementation-defined.

4.13.3 Options

The `cp` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−f` If a file descriptor for a destination file cannot be obtained, as 2
described in step (3a)[2], attempt to unlink the destination file 2
and proceed. 2

`−i` Write a prompt to standard error before copying to any existing destination file. If the response from the standard input is affirmative, the copy shall be attempted, otherwise not.

`−p` Duplicate the following characteristics of each source file in the corresponding destination file:

(1) The time of last data modification and time of last access. If this duplication fails for any reason, `cp` shall write a diagnostic message to standard error.

(2) The user ID and group ID. If this duplication fails for any reason, it is unspecified whether `cp` writes a diagnostic message to standard error.

(3) The file permission bits and the S_ISUID and S_ISGID bits. Other, implementation-defined, bits may be duplicated as well. If this duplication fails for any reason, `cp` shall write a diagnostic message to standard error.

If the user ID or the group ID cannot be duplicated, the file permission bits S_ISUID and S_ISGID shall be cleared. If these bits are present in the source file but are not duplicated in the destination file, it is unspecified whether `cp` writes a diagnostic message to standard error.

The order in which the preceding characteristics are duplicated is unspecified. The *dest_file* shall not be deleted if these characteristics cannot be preserved.

`−R` Copy file hierarchies.

`−r` Copy file hierarchies. The treatment of special files is implemen- 1
tation defined. 1

4.13.4 Operands

The following operands shall be supported by the implementation:

source_file A pathname of a file to be copied.

target_file A pathname of an existing or nonexisting file, used for the output when a single file is copied.

target A pathname of a directory to contain the copied file(s).

4.13.5 External Influences

4.13.5.1 Standard Input

Used to read an input line in response to each prompt specified in Standard Error. Otherwise, the standard input shall not be used.

4.13.5.2 Input Files

The input files specified as operands may be of any file type.

4.13.5.3 Environment Variables

The following environment variables shall affect the execution of `cp`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements used in the extended regular expression defined for the `yesexpr` locale keyword in the LC_MESSAGES category.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior of character classes used in the extended regular expression defined for the `yesexpr` locale keyword in the LC_MESSAGES category.

LC_MESSAGES This variable shall determine the processing of affirmative responses and the language in which messages should be written.

4.13.5.4 Asynchronous Events

Default.

4.13.6 External Effects

4.13.6.1 Standard Output

None.

4.13.6.2 Standard Error

A prompt shall be written to standard error under the conditions specified in 4.13.2. The prompt shall contain the destination pathname, but its format is otherwise unspecified. Otherwise, the standard error shall be used only for diagnostic messages.

4.13.6.3 Output Files

The output files may be of any type.

4.13.7 Extended Description

None.

4.13.8 Exit Status

The `cp` utility shall exit with one of the following values:

0	No error occurred.
>0	An error occurred.

4.13.9 Consequences of Errors

If `cp` is prematurely terminated by a signal or error, files or file hierarchies may be only partially copied and files and directories may have incorrect permissions or access and modification times.

4.13.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

None.

History of Decisions Made

2

The −i option exists on BSD systems, giving applications and users a way to avoid accidentally removing files when copying. Although the 4.3BSD version does not prompt if the standard input is not a terminal, the working group decided that use of −i is a request for interaction, so when the destination path exists, the utility takes instructions from whatever responds on standard input.

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified, because implementations may desire more descriptive prompts than those used on historical implementations. Therefore, an application using the −i option relies on the system to provide the most suitable dialogue directly with the user, based on the behavior specified.

The −p option is historical practice on BSD systems, duplicating the time of last data modification and time of last access. POSIX.2 extends it to preserve the user and group IDs, as well as the file permissions. This requirement has obvious problems in that the directories are almost certainly modified after being copied. This specification requires that the modification times be preserved even so. The statement that the order in which the characteristics are duplicated is unspecified is to permit implementations to provide the maximum amount of security for the user. Implementations should take into account the obvious security issues involved in setting the owner, group, and mode in the wrong order or creating files with an owner, group, or mode different from the final value.

It is unspecified whether `cp` writes diagnostic messages when the user and group IDs cannot be set due to the widespread practice of users using −p to duplicate some portion of the file characteristics, indifferent to the duplication of others. Historic implementations only write diagnostic messages on errors other than [EPERM].

The −r option is historical practice on BSD and BSD-derived systems, copying file hierarchies as opposed to single files. This functionality is used heavily in existing applications and its loss would significantly decrease consensus. The −R option was added as a close synonym to the −r option, selected for consistency with all other options in the standard that do recursive directory descent.

The difference between −R and −r is in the treatment by `cp` of file types other than regular and directory. The original −r flag, for historic reasons, does not handle special files any differently than regular files, but always reads the file and copies its contents. This has obvious problems in the presence of special file types, for example character devices, FIFOs, and sockets. The current `cp` utility specification is intended to require that the −R option recreate the file hierarchy and that the −r option support historical practice. It is anticipated that a future version of this standard will deprecate the `−r` option, and for that reason, there has been no attempt to fix its behavior with respect to FIFOs or other file types where copying the file is clearly wrong. However, some systems support `−r` with the same abilities as the `−R` defined in POSIX.2. To accommodate them as well as systems that do not, the differences between `−r` and `−R` are implementation defined. Implementations may make them identical.

When a failure occurs during the copying of a file hierarchy, `cp` is required to attempt to copy files that are on the same level in the hierarchy or above the file where the failure occurred. It is unspecified if `cp` shall attempt to copy files below the file where the failure occurred (which cannot succeed in any case).

Permissions, owners, and groups of created special file types have been deliberately left as implementation defined. This is to allow systems to satisfy special requirements (for example, allowing users to create character special devices, but requiring them to be owned by a certain group). In general, it is strongly suggested that the permissions, owner, and group be the same as if the user had run the traditional `mknod`, `ln`, or other utility to create the file. It is also probable that additional privileges will be required to create block, character, or other, implementation-specific, special file types.

Additionally, the `−p` option explicitly requires that all set-user-ID and set-group-ID permissions be discarded if any of the owner or group IDs cannot be set. This is to keep users from unintentionally giving away special privilege when copying programs.

When creating regular files, historical versions of `cp` use the mode of the source file as modified by the file mode creation mask. Other choices would have been to use the mode of the source file unmodified by the creation mask, or to use the same mode as would be given to a new file created by the user, plus the execution bits of the source file, and then modified by the file mode creation mask. In the absence of any strong reason to change historic practice, it was in large part retained.

The one difference is that the set-user-ID and set-group-ID bits are explicitly cleared when files are created. This is to prevent users from creating programs that are set-user-ID/set-group-ID to them when copying files or to make set-user-ID/set-group-ID files accessible to new groups of users. For example, if a file is set-user-ID and the copy has a different group ID than the source, a new group of users have execute permission to a set-user-ID program than did previously. In particular, this is a problem for super-users copying users' trees. A finer granularity of protection could be specified, in that the set-user-ID/set-group-ID bits could be retained under certain conditions even if the owner or group could not be set, based on a determination that no additional privileges were provided to any users. This was not seen as sufficiently useful for the added complexity.

When creating directories, historical versions of `cp` use the mode of the source directory, plus read, write, and search bits for the owner, as modified by the file mode creation mask. This is done so that `cp` can copy trees where the user has read permission, but the owner does not. A side effect is that if the file creation mask denies the owner permissions, `cp` will fail. Also, once the copy is done, historical versions of `cp` set the permissions on the created directory to be the same as the source directory, unmodified by the file creation mask.

This behavior has been modified so that `cp` will always be able to create the contents of the directory, regardless of the file creation mask. After the copy is done, the permissions are set to be the same as the source directory, as modified by the file creation mask. This latter change from historical behavior is to prevent users from accidentally creating directories with permissions beyond those they would normally set and for consistency with the behavior of `cp` in creating files.

It is not a requirement that `cp` detect attempts to copy a file to itself; however, implementations are strongly encouraged to do so. Historical implementations have detected the attempt in most cases, which is probably all that is needed.

There are two methods of copying subtrees in this standard. The other method is described as part of the `pax` utility (see 4.48). Both methods are historical practice. The `cp` utility provides a simpler, more intuitive interface, while `pax` offers a finer granularity of control. Each provides additional functionality to the other; in particular, `pax` maintains the hard-link structure of the hierarchy, while `cp` does not. It is the intention of the working group that the results be similar (using appropriate option combinations in both utilities). The results are not required to be identical; there seemed insufficient gain to applications to balance the difficulty of implementations having to guarantee that the results would be exactly identical.

The wording allowing `cp` to copy a directory to implementation-defined file types not specified by POSIX.1 {8} is provided so that implementations supporting symbolic links are not required to prohibit copying directories to symbolic links. Other extensions to POSIX.1 {8} file types may need to use this loophole as well.

4.14 `cut` — Cut out selected fields of each line of a file

4.14.1 Synopsis

`cut –b` *list* `[–n]` [*file* . . .]

`cut –c` *list* [*file* . . .]

`cut –f` *list* `[–d` *delim* `]` `[–s]` [*file* . . .]

4.14.2 Description

The `cut` utility shall cut out bytes (`–b` option), characters (`–c` option), or character-delimited fields (`–f` option) from each line in one or more files, concatenate them, and write them to standard output.

4.14.3 Options

The `cut` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The option-argument *list* (see options `–b`, `–c`, and `–f` below) shall be a comma- 2
separated list or `<blank>`-separated list of positive numbers and ranges. Ranges 2
can be in three forms. The first is two positive numbers separated by a hyphen (*low*–*high*), which represents all fields from the first number to the second number. The second is a positive number preceded by a hyphen (–*high*), which represents all fields from field number 1 to that number. The third is a positive number followed by a hyphen (*low*–), which represents that number to the last field, inclusive. The elements in list can be repeated, can overlap, and can be specified in any order.

The following options shall be supported by the implementation:

`–b` *list* Cut based on a *list* of bytes. Each selected byte shall be output unless the `–n` option is also specified. It shall not be an error to select bytes not present in the input line.

`–c` *list* Cut based on a *list* of characters. Each selected character shall be output. It shall not be an error to select characters not present in the input line.

`–d` *delim* Set the field delimiter to the character *delim* . The default is the `<tab>` character.

`–f` *list* Cut based on a *list* of fields, assumed to be separated in the file by a delimiter character (see `–d`). Each selected field shall be output. Output fields shall be separated by a single occurrence of the field delimiter character. Lines with no field delimiters shall be passed through intact, unless `–s` is specified. It shall not be an error to select fields not present in the input line.