



Linux Ubuntu 26.04 Manual Pages on command 'readlink.2'

\$ man readlink.2

readlink(2) System Calls Manual readlink(2)

NAME

readlink, readlinkat - read value of a symbolic link

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <unistd.h>

ssize_t readlink(size_t bufsiz;
                 const char *restrict path,
                 char buf[restrict bufsiz], size_t bufsiz);

#include <fcntl.h>      /* Definition of AT_* constants */
#include <unistd.h>

ssize_t readlinkat(size_t bufsiz;
                  int dirfd, const char *restrict path,
                  char buf[restrict bufsiz], size_t bufsiz);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
readlink():
    _XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200112L
    || /* glibc <= 2.19: */ _BSD_SOURCE
```

readlinkat():

Since glibc 2.10:

```
    _POSIX_C_SOURCE >= 200809L
```

Before glibc 2.10:

`_ATFILE_SOURCE`

DESCRIPTION

`readlink()` places the contents of the symbolic link path in the buffer `buf`, which has size `bufsiz`. `readlink()` does not append a terminating null byte to `buf`. It will (silently) truncate the contents (to a length of `bufsiz` characters), in case the buffer is too small to hold all of the contents.

`readlinkat()`

The `readlinkat()` system call operates in exactly the same way as `readlink()`, except for the differences described here.

If path is relative, then it is interpreted relative to the directory referred to by the file descriptor `dirfd` (rather than relative to the current working directory of the calling process, as is done by `readlink()` for a relative pathname).

If path is relative and `dirfd` is the special value `AT_FDCWD`, then path is interpreted relative to the current working directory of the calling process (like `readlink()`).

If path is absolute, then `dirfd` is ignored.

Since Linux 2.6.39, path can be an empty string, in which case the call operates on the symbolic link referred to by `dirfd` (which should have been obtained using `open(2)` with the `O_PATH` and `O_NOFOLLOW` flags).

See `openat(2)` for an explanation of the need for `readlinkat()`.

RETURN VALUE

On success, these calls return the number of bytes placed in `buf`. (If the returned value equals `bufsiz`, then truncation may have occurred.) On error, -1 is returned and `errno` is set to indicate the error.

ERRORS

EACCES Search permission is denied for a component of the path prefix. (See also `path_resolution(7)`.)

EBADF (`readlinkat()`) path is relative but `dirfd` is neither `AT_FDCWD` nor a valid file descriptor.

EFAULT `buf` extends outside the process's allocated address space.

EINVAL `bufsiz` is not positive.

EINVAL The named file (i.e., the final filename component of path) is not a symbolic link.

EIO An I/O error occurred while reading from the filesystem.

ELOOP Too many symbolic links were encountered in translating the pathname.

ENAMETOOLONG

A pathname, or a component of a pathname, was too long.

ENOENT The named file does not exist.

ENOMEM Insufficient kernel memory was available.

ENOTDIR

A component of the path prefix is not a directory.

ENOTDIR

(readlinkat()) path is relative and dirfd is a file descriptor referring to a file other than a directory.

STANDARDS

POSIX.1-2024.

HISTORY

readlink()

4.4BSD (first appeared in 4.2BSD), POSIX.1-2001, POSIX.1-2008.

readlinkat()

POSIX.1-2008. Linux 2.6.16, glibc 2.4.

Up to and including glibc 2.4, the return type of readlink() was declared as int. Nowadays, the return type is declared as ssize_t, as (newly) required in POSIX.1-2001.

glibc

On older kernels where readlinkat() is unavailable, the glibc wrapper function falls back to the use of readlink(). When path is relative, glibc constructs a pathname based on the symbolic link in /proc/self/fd that corresponds to the dirfd argument.

NOTES

Using a statically sized buffer might not provide enough room for the symbolic link contents. The required size for the buffer can be obtained from the stat.st_size value returned by a call to lstat(2) on the link. However, the number of bytes written by readlink() and readlinkat() should be checked to make sure that the size of the symbolic link did not increase between the calls. Dynamically allocating the buffer for readlink() and readlinkat() also addresses a common portability problem when using PATH_MAX for the buffer size, as this constant is not guaranteed to be defined per POSIX if the system does not have such limit.

EXAMPLES

The following program allocates the buffer needed by readlink() dynamically from the malloc()

mation provided by lstat(2), falling back to a buffer of size PATH_MAX in cases where lstat(2) reports a size of zero.

```
#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <unistd.h>

int

main(int argc, char *argv[])

{

    char    *buf;

    ssize_t  nbytes, bufsiz;

    struct stat sb;

    if (argc != 2) {

        fprintf(stderr, "Usage: %s <path>\n", argv[0]);

        exit(EXIT_FAILURE);

    }

    if (lstat(argv[1], &sb) == -1) {

        perror("lstat");

        exit(EXIT_FAILURE);

    }

    /* Add one to the link size, so that we can determine whether

       the buffer returned by readlink() was truncated. */

    bufsiz = sb.st_size + 1;

    /* Some magic symlinks under (for example) /proc and /sys

       report 'st_size' as zero. In that case, take PATH_MAX as

       a "good enough" estimate. */

    if (sb.st_size == 0)

        bufsiz = PATH_MAX;

    buf = malloc(bufsiz);

    if (buf == NULL) {

        perror("malloc");
```

```

    exit(EXIT_FAILURE);
}

nbytes = readlink(argv[1], buf, bufsiz);
if (nbytes == -1) {
    perror("readlink");
    exit(EXIT_FAILURE);
}

/* Print only 'nbytes' of 'buf', as it doesn't contain a terminating
   null byte ('\0'). */
printf("%s' points to '%s'\n", argv[1], (int) nbytes, buf);

/* If the return value was equal to the buffer size, then
   the link target was larger than expected (perhaps because the
   target was changed between the call to lstat() and the call to
   readlink()). Warn the user that the returned target may have
   been truncated. */
if (nbytes == bufsiz)
    printf("(Returned buffer may have been truncated)\n");

free(buf);

exit(EXIT_SUCCESS);
}

```

SEE ALSO

readlink(1), lstat(2), stat(2), symlink(2), realpath(3), path_resolution(7), symlink(7)