Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'access.3p' command

$ man access.3p

ACCESS(3P) POSIX Programmer's Manual ACCESS(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

access, faccessat ? determine accessibility of a file descriptor

SYNOPSIS

```
#include <unistd.h>
int access(const char *path, int amode);
#include <fcntl.h>
int faccessat(int fd, const char *path, int amode, int flag);
```

DESCRIPTION

The access() function shall check the file named by the pathname pointed to by the path argument for accessibility according to the bit pattern contained in amode. The checks for accessibility (including directory permissions checked during pathname resolution) shall be per? formed using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID.

The value of amode is either the bitwise-inclusive OR of the access permissions to be checked (R_OK, W_OK, X_OK) or the existence test (F_OK).

If any access permissions are checked, each shall be checked individu?
ally, as described in the Base Definitions volume of POSIX.1?2017, Sec?
tion 4.5, File Access Permissions, except that where that description
refers to execute permission for a process with appropriate privileges,
an implementation may indicate success for X_OK even if execute permis?
sion is not granted to any user.

The faccessat() function, when called with a flag value of zero, shall
be equivalent to the access() function, except in the case where path
specifies a relative path. In this case the file whose accessibility is
to be determined shall be located relative to the directory associated
with the file descriptor fd instead of the current working directory.

If the access mode of the open file description associated with the
file descriptor is not O_SEARCH, the function shall check whether di?
rectory searches are permitted using the current permissions of the di?
rectory underlying the file descriptor. If the access mode is
O_SEARCH, the function shall not perform the check.

If faccessat() is passed the special value AT_FDCWD in the fd parame?
ter, the current working directory shall be used and, if flag is zero,
the behavior shall be identical to a call to access().

Values for flag are constructed by a bitwise-inclusive OR of flags from
the following list, defined in <fcntl.h>:

AT_EACCESS The checks for accessibility (including directory permis?
sions checked during pathname resolution) shall be per?
formed using the effective user ID and group ID instead of
the real user ID and group ID as required in a call to ac?
cess().

RETURN VALUE

Upon successful completion, these functions shall return 0. Otherwise,
these functions shall return -1 and set errno to indicate the error.

ERRORS

These functions shall fail if:

EACCES Permission bits of the file mode do not permit the requested ac?
cess, or search permission is denied on a component of the path

prefix.

ELOOP A loop exists in symbolic links encountered during resolution of the path argument.

ENAMETOOLONG

The length of a component of a pathname is longer than {NAME_MAX}.

ENOENT A component of path does not name an existing file or path is an empty string.

ENOTDIR

A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory, or the path argument contains at least one non-<slash> character and ends with one or more trailing <slash> characters and the last pathname component names an existing file that is neither a di? rectory nor a symbolic link to a directory.

EROFS Write access is requested for a file on a read-only file system.

The faccessat() function shall fail if:

EACCES The access mode of the open file description associated with fd is not O_SEARCH and the permissions of the directory underlying fd do not permit directory searches.

EBADF The path argument does not specify an absolute path and the fd argument is neither AT_FDCWD nor a valid file descriptor open for reading or searching.

ENOTDIR

The path argument is not an absolute path and fd is a file de? scriptor associated with a non-directory file.

These functions may fail if:

EINVAL The value of the amode argument is invalid.

ELOOP More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the path argument.

ENAMETOOLONG

The length of a pathname exceeds {PATH_MAX}, or pathname resolu? tion of a symbolic link produced an intermediate result with a

length that exceeds {PATH_MAX}.

ETXTBSY

Write access is requested for a pure procedure (shared text) file that is being executed.

The faccessat() function may fail if:

EINVAL The value of the flag argument is not valid.

The following sections are informative.

EXAMPLES

Testing for the Existence of a File

The following example tests whether a file named myfile exists in the /tmp directory.

```
#include <unistd.h>
...
int result;
const char *pathname = "/tmp/myfile";
result = access (pathname, F_OK);
```

APPLICATION USAGE

Use of these functions is discouraged since by the time the returned information is acted upon, it is out-of-date. (That is, acting upon the information always leads to a time-of-check-to-time-of-use race condi? tion.) An application should instead attempt the action itself and han? dle the [EACCES] error that occurs if the file is not accessible (with a change of effective user and group IDs beforehand, and perhaps a change back afterwards, in the case where access() or faccessat() with? out AT_EACCES would have been used.)

Historically, one of the uses of access() was in set-user-ID root pro? grams to check whether the user running the program had access to a file. This relied on ``super-user'' privileges which were granted based on the effective user ID being zero, so that when access() used the real user ID to check accessibility those privileges were not taken into account. On newer systems where privileges can be assigned which have no association with user or group IDs, if a program with such privileges calls access(), the change of IDs has no effect on the priv?

ileges and therefore they are taken into account in the accessibility checks. Thus, access() (and faccessat() with flag zero) cannot be used for this historical purpose in such programs. Likewise, if a system provides any additional or alternate file access control mechanisms that are not user ID-based, they will still be taken into account.

If a relative pathname is used, no account is taken of whether the cur? rent directory (or the directory associated with the file descriptor fd) is accessible via any absolute pathname. Applications using ac? cess(), or faccessat() without AT_EACCES, may consequently act as if the file would be accessible to a user with the real user ID and group ID of the process when such a user would not in practice be able to ac? cess the file because access would be denied at some point above the current directory (or the directory associated with the file descriptor fd) in the file hierarchy.

If access() or faccessat() is used with W_OK to check for write access to a directory which has the S_ISVTX bit set, a return value indicating the directory is writable can be misleading since some operations on files in the directory would not be permitted based on the ownership of those files (see the Base Definitions volume of POSIX.1?2017, Section 4.3, Directory Protection).

Additional values of amode other than the set defined in the descrip? tion may be valid; for example, if a system has extended access con? trols.

The use of the AT_EACCESS value for flag enables functionality not available in access().

RATIONALE

In early proposals, some inadequacies in the access() function led to the creation of an eaccess() function because:

1. Historical implementations of access() do not test file access cor? rectly when the process' real user ID is superuser. In particular, they always return zero when testing execute permissions without regard to whether the file is executable.
2. The superuser has complete access to all files on a system. As a

consequence, programs started by the superuser and switched to the effective user ID with lesser privileges cannot use access() to test their file access permissions.

However, the historical model of eaccess() does not resolve problem (1), so this volume of POSIX.1?2017 now allows access() to behave in the desired way because several implementations have corrected the problem. It was also argued that problem (2) is more easily solved by using open(), chdir(), or one of the exec functions as appropriate and responding to the error, rather than creating a new function that would not be as reliable. Therefore, eaccess() is not included in this volume of POSIX.1?2017.

The sentence concerning appropriate privileges and execute permission bits reflects the two possibilities implemented by historical implemen? tations when checking superuser access for X_OK.

New implementations are discouraged from returning X_OK unless at least one execution permission bit is set.

The purpose of the faccessat() function is to enable the checking of the accessibility of files in directories other than the current work? ing directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to access(), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the faccessat() function it can be guaranteed that the file tested for accessibility is located relative to the desired directory.

FUTURE DIRECTIONS

These functions may be formally deprecated (for example, by shading them OB) in a future version of this standard.

SEE ALSO

chmod(), fstatat()

The Base Definitions volume of POSIX.1?2017, Section 4.5, File Access Permissions, <fcntl.h>, <unistd.h>

COPYRIGHT