



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

\$ man open_by_handle_at.2

open_by_handle_at(2) System Calls Manual open_by_handle_at(2)

NAME

name_to_handle_at, open_by_handle_at - obtain handle for a pathname and open file via a handle

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#define _GNU_SOURCE            /* See feature_test_macros(7) */
#include <fcntl.h>

int name_to_handle_at(int dirfd, const char *path,
                     struct file_handle *handle,
                     int *mount_id, int flags);

int open_by_handle_at(int mount_fd, struct file_handle *handle,
                     int flags);
```

DESCRIPTION

The name_to_handle_at() and open_by_handle_at() system calls split the functionality of open(2) into two parts: name_to_handle_at() returns an opaque handle that corresponds to a specified file; open_by_handle_at() opens the file corresponding to a handle returned by a previous call to name_to_handle_at() and returns an open file descriptor.

name_to_handle_at()

The name_to_handle_at() system call returns a file handle and a mount ID corresponding to the file specified by the dirfd and path arguments. The file handle is returned via the ar?

gument handle, which is a pointer to a structure of the following form:

```

struct file_handle {
    unsigned int  handle_bytes; /* Size of f_handle [in, out] */
    int          handle_type; /* Handle type [out] */
    unsigned char f_handle[0]; /* File identifier (sized by
                                caller) [out] */
};

```

It is the caller's responsibility to allocate the structure with a size large enough to hold the handle returned in `f_handle`. Before the call, the `handle_bytes` field should be initialized to contain the allocated size for `f_handle`. (The constant `MAX_HANDLE_SZ`, defined in `<fcntl.h>`, specifies the maximum expected size for a file handle. It is not a guaranteed upper limit as future filesystems may require more space.) Upon successful return, the `handle_bytes` field is updated to contain the number of bytes actually written to `f_handle`. The caller can discover the required size for the `file_handle` structure by making a call in which `handle->handle_bytes` is zero; in this case, the call fails with the error `EOVERFLOW` and `handle->handle_bytes` is set to indicate the required size; the caller can then use this information to allocate a structure of the correct size (see EXAMPLES below). Some care is needed here as `EOVERFLOW` can also indicate that no file handle is available for this particular name in a filesystem which does normally support file-handle lookup. This case can be detected when the `EOVERFLOW` error is returned without `handle_bytes` being increased. Other than the use of the `handle_bytes` field, the caller should treat the `file_handle` structure as an opaque data type: the `handle_type` and `f_handle` fields can be used in a subsequent call to `open_by_handle_at()`. The caller can also use the opaque `file_handle` to compare the identity of filesystem objects that were queried at different times and possibly at different paths. The fanotify(7) subsystem can report events with an information record containing a `file_handle` to identify the filesystem object.

The flags argument is a bit mask constructed by ORing together zero or more of `AT_HANDLE_FID`,

`AT_HANDLE_MNT_ID_UNIQUE`, `AT_HANDLE_CONNECTABLE`, `AT_EMPTY_PATH`, and `AT_SYMLINK_FOLLOWS`.

LOW, described below.

When flags contain the `AT_HANDLE_FID` (since Linux 6.5) flag, the caller indicates that the returned `file_handle` is needed to identify the filesystem object, and not for opening the file later, so it should be expected that a subsequent call to `open_by_handle_at()` with the returned `file_handle` may fail.

When flags contain the `AT_HANDLE_MNT_ID_UNIQUE` (since Linux 6.12) flag, the caller indicates that the width of the `mount_id` buffer is at least 64 bits, and then the mount id returned in that buffer is the unique mount id as the one returned by `statx(2)` with the `STATX_MNT_ID_UNIQUE` flag.

When flags contain the `AT_HANDLE_CONNECTABLE` (since Linux 6.13) flag, the caller indicates that the returned `file_handle` is needed to open a file with known path later, so it should be expected that a subsequent call to `open_by_handle_at()` with the returned `file_handle` may fail if the file was moved, but otherwise, the path of the opened file is expected to be visible from the `/proc/pid/fd/*` magic link. This flag can not be used in combination with the flags `AT_HANDLE_FID` and/or `AT_EMPTY_PATH`.

Together, the `path` and `dirfd` arguments identify the file for which a handle is to be obtained. There are four distinct cases:

- ? If `path` is a nonempty string containing an absolute pathname, then a handle is returned for the file referred to by that pathname. In this case, `dirfd` is ignored.
- ? If `path` is a nonempty string containing a relative pathname and `dirfd` has the special value `AT_FDCWD`, then `path` is interpreted relative to the current working directory of the caller, and a handle is returned for the file to which it refers.
- ? If `path` is a nonempty string containing a relative pathname and `dirfd` is a file descriptor referring to a directory, then `path` is interpreted relative to the directory referred to by `dirfd`, and a handle is returned for the file to which it refers. (See `openat(2)` for an explanation of why "directory file descriptors" are useful.)
- ? If `path` is an empty string and flags specifies the value `AT_EMPTY_PATH`, then `dirfd` can be an open file descriptor referring to any type of file, or `AT_FDCWD`, meaning the current working directory, and a handle is returned for the file to which it refers.

The `mount_id` argument returns an identifier for the filesystem mount that corresponds to `path`. This corresponds to the first field in one of the records in `/proc/self/mountinfo`.

Opening the `pathname` in the fifth field of that record yields a file descriptor for the mount point; that file descriptor can be used in a subsequent call to `open_by_handle_at()`. `mount_id` is returned both for a successful call and for a call that results in the error `E_OVERFLOW`.

By default, `name_to_handle_at()` does not dereference `path` if it is a symbolic link, and thus returns a handle for the link itself. If `AT_SYMLINK_FOLLOW` is specified in flags, `path` is dereferenced if it is a symbolic link (so that the call returns a handle for the file re?

ferred to by the link).

`name_to_handle_at()` does not trigger a mount when the final component of the pathname is an automount point. When a filesystem supports both file handles and automount points, a `name_to_handle_at()` call on an automount point will return with error `EOVERFLOW` without having increased `handle_bytes`. This can happen since Linux 4.13 with NFS when accessing a directory which is on a separate filesystem on the server. In this case, the automount can be triggered by adding a "/" to the end of the pathname.

`open_by_handle_at()`

The `open_by_handle_at()` system call opens the file referred to by `handle`, a file handle returned by a previous call to `name_to_handle_at()`.

The `mount_fd` argument is a file descriptor for any object (file, directory, etc.) in the mounted filesystem with respect to which `handle` should be interpreted. The special value `AT_FDCWD` can be specified, meaning the current working directory of the caller.

The `flags` argument is as for `open(2)`. If `handle` refers to a symbolic link, the caller must specify the `O_PATH` flag, and the symbolic link is not dereferenced; the `O_NOFOLLOW` flag, if specified, is ignored.

The caller must have the `CAP_DAC_READ_SEARCH` capability to invoke `open_by_handle_at()`.

RETURN VALUE

On success, `name_to_handle_at()` returns 0, and `open_by_handle_at()` returns a file descriptor (a nonnegative integer).

In the event of an error, both system calls return -1 and set `errno` to indicate the error.

ERRORS

`name_to_handle_at()` and `open_by_handle_at()` can fail for the same errors as `openat(2)`. In addition, they can fail with the errors noted below.

`name_to_handle_at()` can fail with the following errors:

`EFAULT` `path`, `mount_id`, or `handle` points outside your accessible address space.

`EINVAL` `flags` includes an invalid bit value or an invalid bit combination.

`EINVAL` `handle->handle_bytes` is greater than `MAX_HANDLE_SZ`.

`ENOENT` `path` is an empty string, but `AT_EMPTY_PATH` was not specified in `flags`.

`ENOTDIR`

The file descriptor supplied in `dirfd` does not refer to a directory, and it is not the case that both `flags` includes `AT_EMPTY_PATH` and `path` is an empty string.

`EOPNOTSUPP`

The filesystem does not support decoding of a pathname to a file handle.

EOVERFLOW

The handle->handle_bytes value passed into the call was too small. When this error occurs, handle->handle_bytes is updated to indicate the required size for the handle.

open_by_handle_at() can fail with the following errors:

EBADF mount_fd is not an open file descriptor.

EBADF path is relative but dirfd is neither AT_FDCWD nor a valid file descriptor.

EFAULT handle points outside your accessible address space.

EINVAL handle->handle_bytes is greater than MAX_HANDLE_SZ or is equal to zero.

ELOOP handle refers to a symbolic link, but O_PATH was not specified in flags.

EPERM The caller does not have the CAP_DAC_READ_SEARCH capability.

ESTALE The specified handle is not valid for opening a file. This error will occur if, for example, the file has been deleted. This error can also occur if the handle was acquired using the AT_HANDLE_FID flag and the filesystem does not support open_by_handle_at(). This error can also occur if the handle was acquired using the AT_HANDLE_DLE_CONNECTABLE flag and the file was moved to a different parent.

VERSIONS

FreeBSD has a broadly similar pair of system calls in the form of getfh() and fhopen().

STANDARDS

Linux.

HISTORY

Linux 2.6.39, glibc 2.14.

NOTES

A file handle can be generated in one process using name_to_handle_at() and later used in a different process that calls open_by_handle_at().

Some filesystems don't support the translation of pathnames to file handles, for example, /proc, /sys, and various network filesystems. Some filesystems support the translation of pathnames to file handles, but do not support using those file handles in open_by_handle_at().

A file handle may become invalid ("stale") if a file is deleted, or for other filesystem-specific reasons. Invalid handles are notified by an ESTALE error from open_by_handle_at().

These system calls are designed for use by user-space file servers. For example, a user-space NFS server might generate a file handle and pass it to an NFS client. Later, when the

client wants to open the file, it could pass the handle back to the server. This sort of functionality allows a user-space file server to operate in a stateless fashion with respect to the files it serves.

If `path` refers to a symbolic link and `flags` does not specify `AT_SYMLINK_FOLLOW`, then `name_to_handle_at()` returns a handle for the link (rather than the file to which it refers).

The process receiving the handle can later perform operations on the symbolic link by converting the handle to a file descriptor using `open_by_handle_at()` with the `O_PATH` flag, and then passing the file descriptor as the `dirfd` argument in system calls such as `readlinkat(2)` and `fchownat(2)`.

Obtaining a persistent filesystem ID

The mount IDs in `/proc/self/mountinfo` can be reused as filesystems are unmounted and mounted. Therefore, the mount ID returned by `name_to_handle_at()` (in `*mount_id`) should not be treated as a persistent identifier for the corresponding mounted filesystem. However, an application can use the information in the `mountinfo` record that corresponds to the mount ID to derive a persistent identifier.

For example, one can use the device name in the fifth field of the `mountinfo` record to search for the corresponding device UUID via the symbolic links in `/dev/disks/by-uuid`. (A more comfortable way of obtaining the UUID is to use the `libblkid(3)` library.) That process can then be reversed, using the UUID to look up the device name, and then obtaining the corresponding mount point, in order to produce the `mount_fd` argument used by `open_by_handle_at()`.

EXAMPLES

The two programs below demonstrate the use of `name_to_handle_at()` and `open_by_handle_at()`.

The first program (`t_name_to_handle_at.c`) uses `name_to_handle_at()` to obtain the file handle and mount ID for the file specified in its command-line argument; the handle and mount ID are written to standard output.

The second program (`t_open_by_handle_at.c`) reads a mount ID and file handle from standard input. The program then employs `open_by_handle_at()` to open the file using that handle. If an optional command-line argument is supplied, then the `mount_fd` argument for `open_by_handle_at()` is obtained by opening the directory named in that argument. Otherwise, `mount_fd` is obtained by scanning `/proc/self/mountinfo` to find a record whose mount ID matches the mount ID read from standard input, and the mount directory specified in that record is opened. (These programs do not deal with the fact that mount IDs are not persistent.)

The following shell session demonstrates the use of these two programs:

```
$ echo 'Can you please think about it?' > cecilia.txt;

$ ./t_name_to_handle_at cecilia.txt > fh;

$ ./t_open_by_handle_at < fh;

open_by_handle_at: Operation not permitted

$ sudo ./t_open_by_handle_at < fh;    # Need CAP_SYS_ADMIN

Read 31 bytes

$ rm cecilia.txt;
```

Now we delete and (quickly) re-create the file so that it has the same content and (by chance) the same inode. Nevertheless, `open_by_handle_at()` recognizes that the original file referred to by the file handle no longer exists.

```
$ stat --printf="%i\n" cecilia.txt;    # Display inode number

4072121

$ rm cecilia.txt;

$ echo 'Can you please think about it?' > cecilia.txt;

$ stat --printf="%i\n" cecilia.txt;    # Check inode number

4072121

$ sudo ./t_open_by_handle_at < fh;

open_by_handle_at: Stale NFS file handle
```

Program source: `t_name_to_handle_at.c`

```
#define _GNU_SOURCE

#include <err.h>

#include <errno.h>

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

int

main(int argc, char *argv[])

{

    int          mount_id, fhsize, flags, dirfd;

    char          *path;

    struct file_handle *fhp;

    if (argc != 2) {
```

```

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

    exit(EXIT_FAILURE);
}

path = argv[1];

/* Allocate file_handle structure. */

fhsz = sizeof(*fhp);

fhp = malloc(fhsz);

if (fhp == NULL)

    err(EXIT_FAILURE, "malloc");

/* Make an initial call to name_to_handle_at() to discover

    the size required for file handle. */

dirfd = AT_FDCWD;          /* For name_to_handle_at() calls */

flags = 0;                 /* For name_to_handle_at() calls */

fhp->handle_bytes = 0;

if (name_to_handle_at(dirfd, path, fhp, &mount_id, flags) != -1

    || errno != EOVERFLOW)

{

    fprintf(stderr, "Unexpected result from name_to_handle_at()\n");

    exit(EXIT_FAILURE);

}

/* Reallocate file_handle structure with correct size. */

fhsz = sizeof(*fhp) + fhp->handle_bytes;

fhp = realloc(fhp, fhsz);    /* Copies fhp->handle_bytes */

if (fhp == NULL)

    err(EXIT_FAILURE, "realloc");

/* Get file handle from pathname supplied on command line. */

if (name_to_handle_at(dirfd, path, fhp, &mount_id, flags) == -1)

    err(EXIT_FAILURE, "name_to_handle_at");

/* Write mount ID, file handle size, and file handle to stdout,

    for later reuse by t_open_by_handle_at.c. */

printf("%d\n", mount_id);

printf("%u %d  ", fhp->handle_bytes, fhp->handle_type);

for (size_t j = 0; j < fhp->handle_bytes; j++)

```

```

        printf(" %02x", fhp->f_handle[j]);

    printf("\n");

    exit(EXIT_SUCCESS);

}

```

Program source: t_open_by_handle_at.c

```

#define _GNU_SOURCE

#include <err.h>

#include <fcntl.h>

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <unistd.h>

/* Scan /proc/self/mountinfo to find the line whose mount ID matches
'mount_id'. (An easier way to do this is to install and use the
'libmount' library provided by the 'util-linux' project.)

Open the corresponding mount path and return the resulting file
descriptor. */

static int

open_mount_path_by_id(int mount_id)

{
    int    mi_mount_id, found;

    char    mount_path[PATH_MAX];

    char    *linep;

    FILE    *fp;

    size_t    lsize;

    ssize_t    nread;

    fp = fopen("/proc/self/mountinfo", "r");

    if (fp == NULL)

        err(EXIT_FAILURE, "fopen");

    found = 0;

    linep = NULL;

```

```

while (!found) {
    nread = getline(&linep, &lsize, fp);

    if (nread == -1)
        break;

    nread = sscanf(linep, "%d %d %s %s %s",
        &mi_mount_id, mount_path);

    if (nread != 2) {
        fprintf(stderr, "Bad sscanf()\n");
        exit(EXIT_FAILURE);
    }

    if (mi_mount_id == mount_id)
        found = 1;
}

free(linep);
fclose(fp);

if (!found) {
    fprintf(stderr, "Could not find mount point\n");
    exit(EXIT_FAILURE);
}

return open(mount_path, O_RDONLY);
}

int
main(int argc, char *argv[])
{
    int          mount_id, fd, mount_fd, handle_bytes;
    char         buf[1000];

#define LINE_SIZE 100
    char         line1[LINE_SIZE], line2[LINE_SIZE];
    char         *nextp;
    ssize_t      nread;
    struct file_handle *fhp;

    if ((argc > 1 && strcmp(argv[1], "--help") == 0) || argc > 2) {
        fprintf(stderr, "Usage: %s [mount-path]\n", argv[0]);

```

```

    exit(EXIT_FAILURE);
}

/* Standard input contains mount ID and file handle information:

    Line 1: <mount_id>

    Line 2: <handle_bytes> <handle_type>  <bytes of handle in hex>

*/

if (fgets(line1, sizeof(line1), stdin) == NULL ||
    fgets(line2, sizeof(line2), stdin) == NULL)
{
    fprintf(stderr, "Missing mount_id / file handle\n");
    exit(EXIT_FAILURE);
}

mount_id = atoi(line1);
handle_bytes = strtoul(line2, &nextp, 0);

/* Given handle_bytes, we can now allocate file_handle structure. */
fhp = malloc(sizeof(*fhp) + handle_bytes);
if (fhp == NULL)
    err(EXIT_FAILURE, "malloc");
fhp->handle_bytes = handle_bytes;
fhp->handle_type = strtoul(nextp, &nextp, 0);
for (size_t j = 0; j < fhp->handle_bytes; j++)
    fhp->f_handle[j] = strtoul(nextp, &nextp, 16);

/* Obtain file descriptor for mount point, either by opening
    the pathname specified on the command line, or by scanning
    /proc/self/mounts to find a mount that matches the 'mount_id'
    that we received from stdin. */

if (argc > 1)
    mount_fd = open(argv[1], O_RDONLY);
else
    mount_fd = open_mount_path_by_id(mount_id);
if (mount_fd == -1)
    err(EXIT_FAILURE, "opening mount fd");

/* Open file using handle and mount point. */

```

```

fd = open_by_handle_at(mount_fd, fhp, O_RDONLY);
if (fd == -1)
    err(EXIT_FAILURE, "open_by_handle_at");
/* Try reading a few bytes from the file. */
nread = read(fd, buf, sizeof(buf));
if (nread == -1)
    err(EXIT_FAILURE, "read");
printf("Read %zd bytes\n", nread);
exit(EXIT_SUCCESS);
}

```

SEE ALSO

open(2), libblkid(3), blkid(8), findfs(8), mount(8)

The libblkid and libmount documentation in the latest util-linux release at <https://www.kernel.org/pub/linux/utils/util-linux/>

Linux man-pages 6.17

2026-02-08

open_by_handle_at(2)