



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

\$ man open_tree.2

open_tree(2) System Calls Manual open_tree(2)

NAME

open_tree - open path or create detached mount object and attach to fd

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#define _GNU_SOURCE      /* See feature_test_macros(7) */
#include <fcntl.h>        /* Definition of AT_* constants */
#include <sys/mount.h>

int open_tree(int dirfd, const char *path, unsigned int flags);

#include <sys/syscall.h>  /* Definition of SYS_* constants */

int syscall(SYS_open_tree_attr,
            int dirfd, const char *path, unsigned int flags,
            struct mount_attr *_Nullable attr, size_t size);
```

Note: glibc provides no wrapper for open_tree_attr(), necessitating the use of syscall(2).

DESCRIPTION

The open_tree() system call is part of the suite of file-descriptor-based mount facilities in Linux.

? If flags contains OPEN_TREE_CLONE, open_tree() creates a detached mount object which consists of a bind-mount of the path specified by the path. A new file descriptor associated with the detached mount object is then returned. The mount object is equivalent to a bind-mount that would be created by mount(2) called with MS_BIND, except that it is tied to a file descriptor and is not mounted onto the filesystem.

As with file descriptors returned from `fsmount(2)`, the resultant file descriptor can then be used with `move_mount(2)`, `mount_setattr(2)`, or other such system calls to do further mount operations.

This mount object will be unmounted and destroyed when the file descriptor is closed if it was not otherwise attached to a mount point by calling `move_mount(2)`. This implicit unmount operation is lazy?akin to calling `umount2(2)` with `MNT_DETACH`; thus, any existing open references to files from the mount object will continue to work, and the mount object will only be completely destroyed once it ceases to be busy.

? If `flags` does not contain `OPEN_TREE_CLONE`, `open_tree()` returns a file descriptor that is exactly equivalent to one produced by `openat(2)` when called with the same `dirfd` and `path`.

In either case, the resultant file descriptor acts the same as one produced by `open(2)` with `O_PATH`, meaning it can also be used as a `dirfd` argument to `"*at()"` system calls. However, unlike `open(2)` called with `O_PATH`, automounts will by default be triggered by `open_tree()` unless `AT_NO_AUTOMOUNT` is included in `flags`.

As with `"*at()"` system calls, `open_tree()` uses the `dirfd` argument in conjunction with the `path` argument to determine the path to operate on, as follows:

? If the `pathname` given in `path` is absolute, then `dirfd` is ignored.

? If the `pathname` given in `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 `open(2)`).

? If the `pathname` given in `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 `open(2)` for a relative `pathname`). In this case, `dirfd` must be a directory that was opened for reading (`O_RDONLY`) or using the `O_PATH` flag.

? If `path` is an empty string, and `flags` contains `AT_EMPTY_PATH`, then the file descriptor `dirfd` is operated on directly. In this case, `dirfd` may refer to any type of file, not just a directory.

See `openat(2)` for an explanation of why the `dirfd` argument is useful.

`flags` can be used to control aspects of the path lookup and properties of the returned file descriptor. A value for `flags` is constructed by bitwise ORing zero or more of the following constants:

`AT_EMPTY_PATH`

If `path` is an empty string, operate on the file referred to by `dirfd` (which may have been obtained from `open(2)`, `fsmount(2)`, or from another `open_tree()` call). In this case, `dirfd` may refer to any type of file, not just a directory. If `dirfd` is `AT_FDCWD`, `open_tree()` will operate on the current working directory of the calling process. This flag is Linux-specific; define `_GNU_SOURCE` to obtain its definition.

`AT_NO_AUTOMOUNT`

Do not automount the terminal ("basename") component of `path` if it is a directory that is an automount point. This allows you to create a handle to the automount point itself, rather than the location it would mount. This flag has no effect if the mount point has already been mounted over. This flag is Linux-specific; define `_GNU_SOURCE` to obtain its definition.

`AT_SYMLINK_NOFOLLOW`

If `path` is a symbolic link, do not dereference it; instead, create either a handle to the link itself or a bind-mount of it. The resultant file descriptor is indistinguishable from one produced by `openat(2)` with `O_PATH|O_NOFOLLOW`.

`OPEN_TREE_CLOEXEC`

Set the close-on-exec (`FD_CLOEXEC`) flag on the new file descriptor. See the description of the `O_CLOEXEC` flag in `open(2)` for reasons why this may be useful.

`OPEN_TREE_CLONE`

Rather than creating an `openat(2)`-style `O_PATH` file descriptor, create a bind-mount of `path` (akin to `mount --bind`) as a detached mount object. In order to do this operation, the calling process must have the `CAP_SYS_ADMIN` capability.

`AT_RECURSIVE`

Create a recursive bind-mount of the `path` (akin to `mount --rbind`) as a detached mount object. This flag is only permitted in conjunction with `OPEN_TREE_CLONE`.

`open_tree_attr()`

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

After performing the same operation as with `open_tree()`, `open_tree_attr()` will apply the

mount attribute changes described in `attr` to the file descriptor before it is returned. (See `mount_attr(2)` for a description of the `mount_attr` structure. As described in `mount_setattr(2)`, `size` must be set to `sizeof(struct mount_attr)` in order to support future extensions.) If `attr` is `NULL`, or has `attr.attr_clr`, `attr.attr_set`, and `attr.propagation` all set to zero, then `open_tree_attr()` has identical behavior to `open_tree()`.

The application of `attr` to the resultant file descriptor has identical semantics to `mount_setattr(2)`, except for the following extensions and general caveats:

? Unlike `mount_setattr(2)` called with a regular `OPEN_TREE_CLONE` detached mount object from `open_tree()`, `open_tree_attr()` can specify a different setting for `MOUNT_ATTR_IDMAP` to the original mount object cloned with `OPEN_TREE_CLONE`.

Adding `MOUNT_ATTR_IDMAP` to `attr.attr_clr` will disable ID-mapping for the new mount object; adding `MOUNT_ATTR_IDMAP` to `attr.attr_set` will configure the mount object to have the ID-mapping defined by the user namespace referenced by the file descriptor `attr.usersns_fd`. (The semantics of which are identical to when `mount_setattr(2)` is used to configure `MOUNT_ATTR_IDMAP`.)

Changing or removing the mapping of an ID-mapped mount is only permitted if a new detached mount object is being created with flags including `OPEN_TREE_CLONE`.

? If flags contains `AT_RECURSIVE`, then the attributes described in `attr` are applied recursively (just as when `mount_setattr(2)` is called with `AT_RECURSIVE`). However, this applies in addition to the `open_tree()`-specific behavior regarding `AT_RECURSIVE`, and thus flags must also contain `OPEN_TREE_CLONE`.

Note that if flags does not contain `OPEN_TREE_CLONE`, `open_tree_attr()` will attempt to modify the mount attributes of the mount object attached at the path described by `dirfd` and `path`.

As with `mount_setattr(2)`, if said path is not a mount point, `open_tree_attr()` will return an error.

RETURN VALUE

On success, a new file descriptor is returned. On error, `-1` is returned, and `errno` is set to indicate the error.

ERRORS

`EACCES` Search permission is denied for one of the directories in the path prefix of `path`.

(See also `path_resolution(7)`.)

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

`EFAULT` `path` is `NULL` or a pointer to a location outside the calling process's accessible address space.

dress space.

EINVAL Invalid flag specified in flags.

ELOOP Too many symbolic links encountered when resolving path.

EMFILE The calling process has too many open files to create more.

ENAMETOOLONG

path is longer than PATH_MAX.

ENFILE The system has too many open files to create more.

ENOENT A component of path does not exist, or is a dangling symbolic link.

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

ENOTDIR

A component of the path prefix of path is not a directory, or path is relative and dirfd is a file descriptor referring to a file other than a directory.

ENOSPC The "anonymous" mount namespace necessary to contain the OPEN_TREE_CLONE detached bind-mount mount object could not be allocated, as doing so would exceed the configured per-user limit on the number of mount namespaces in the current user namespace.
(See also namespaces(7).)

ENOMEM The kernel could not allocate sufficient memory to complete the operation.

EPERM flags contains OPEN_TREE_CLONE but the calling process does not have the required CAP_SYS_ADMIN capability.

STANDARDS

Linux.

HISTORY

open_tree()

Linux 5.2. glibc 2.36.

open_tree_attr()

Linux 6.15.

NOTES

Mount propagation

The bind-mount mount objects created by open_tree() with OPEN_TREE_CLONE are not associated with the mount namespace of the calling process. Instead, each mount object is placed in a newly allocated "anonymous" mount namespace associated with the calling process.

One of the side-effects of this is that (unlike bind-mounts created with mount(2)), mount propagation (as described in mount_namespaces(7)) will not be applied to bind-mounts created

by `open_tree()` until the bind-mount is attached with `move_mount(2)`, at which point the mount object will be associated with the mount namespace where it was attached and mount propagation will resume. Note that any mount propagation events that occurred before the mount object was attached will not be propagated to the mount object, even after it is attached.

EXAMPLES

The following examples show how `open_tree()` can be used in place of more traditional `mount(2)` calls with `MS_BIND`.

```
int srcfd = open_tree(AT_FDCWD, "/var", OPEN_TREE_CLONE);

move_mount(srcfd, "", AT_FDCWD, "/mnt", MOVE_MOUNT_F_EMPTY_PATH);
```

First, a detached bind-mount mount object of `/var` is created and associated with the file descriptor `srcfd`. Then, the mount object is attached to `/mnt` using `move_mount(2)` with `MOVE_MOUNT_F_EMPTY_PATH` to request that the detached mount object associated with the file descriptor `srcfd` be moved (and thus attached) to `/mnt`.

The above procedure is functionally equivalent to the following mount operation using `mount(2)`:

```
mount("/var", "/mnt", NULL, MS_BIND, NULL);
```

`OPEN_TREE_CLONE` can be combined with `AT_RECURSIVE` to create recursive detached bind-mount objects, which in turn can be attached to mount points to create recursive bind-mounts.

```
int srcfd = open_tree(AT_FDCWD, "/var",
                     OPEN_TREE_CLONE | AT_RECURSIVE);

move_mount(srcfd, "", AT_FDCWD, "/mnt", MOVE_MOUNT_F_EMPTY_PATH);
```

The above procedure is functionally equivalent to the following mount operation using `mount(2)`:

```
mount("/var", "/mnt", NULL, MS_BIND | MS_REC, NULL);
```

One of the primary benefits of using `open_tree()` and `move_mount(2)` over the traditional `mount(2)` is that operating with `dirfd`-style file descriptors is far easier and more intuitive.

```
int srcfd = open_tree(100, "", AT_EMPTY_PATH | OPEN_TREE_CLONE);

move_mount(srcfd, "", 200, "foo", MOVE_MOUNT_F_EMPTY_PATH);
```

The above procedure is roughly equivalent to the following mount operation using `mount(2)`:

```
mount("/proc/self/fd/100",
      "/proc/self/fd/200/foo",
```

```
NULL, MS_BIND, NULL);
```

In addition, you can use the file descriptor returned by `open_tree()` as the `dirfd` argument to any `"*at()"` system calls:

```
int dirfd, fd;

dirfd = open_tree(AT_FDCWD, "/etc", OPEN_TREE_CLONE);
fd = openat(dirfd, "passwd", O_RDONLY);
fchmodat(dirfd, "shadow", 0000, 0);
close(dirfd);
close(fd);

/* The bind-mount is now destroyed */
```

`open_tree_attr()`

The following is an example of how `open_tree_attr()` can be used to take an existing id-mapped mount and construct a new bind-mount mount object with a different `MOUNT_ATTR_IDMAP` attribute. The resultant detached mount object can be used like any other mount object returned by `open_tree()`.

```
int nsfd1, nsfd2;
int mntfd1, mntfd2, mntfd3;
struct mount_attr attr;

mntfd1 = open_tree(AT_FDCWD, "/foo", OPEN_TREE_CLONE);

/* Configure the id-mapping of mntfd1 */
nsfd1 = open("/proc/1234/ns/user", O_RDONLY);
memset(&attr, 0, sizeof(attr));
attr.attr_set = MOUNT_ATTR_IDMAP;
attr.usersns_fd = nsfd1;
mount_setattr(mntfd1, "", AT_EMPTY_PATH, &attr, sizeof(attr));

/* Create a new copy with a different id-mapping */
nsfd2 = open("/proc/5678/ns/user", O_RDONLY);
memset(&attr, 0, sizeof(attr));
attr.attr_clr = MOUNT_ATTR_IDMAP;
attr.attr_set = MOUNT_ATTR_IDMAP;
attr.usersns_fd = nsfd2;

mntfd2 = open_tree_attr(mntfd1, "", OPEN_TREE_CLONE,
    &attr, sizeof(attr));
```

```
/* Create a new copy with the id-mapping cleared */
```

```
memset(&attr, 0, sizeof(attr));
```

```
attr.attr_clr = MOUNT_ATTR_IDMAP;
```

```
mntfd3 = open_tree_attr(mntfd1, "", OPEN_TREE_CLONE,  
                        &attr, sizeof(attr));
```

`open_tree_attr()` can also be used with attached mount objects; the above example is only intended to be illustrative.

SEE ALSO

`fsconfig(2)`, `fsmount(2)`, `fsopen(2)`, `fspick(2)`, `mount(2)`, `mount_setattr(2)`, `move_mount(2)`,
`mount_namespaces(7)`

Linux man-pages 6.17

2025-12-25

`open_tree(2)`