mount(2) — Linux manual page

NAME top

mount - mount filesystem

SYNOPSIS top

```
#include <sys/mount.h>

int mount(const char *source, const char *target,
          const char *filesystemtype, unsigned long mountflags,
          const void *data);
```

DESCRIPTION top

mount() attaches the filesystem specified by *source* (which is often a pathname referring to a device, but can also be the pathname of a directory or file, or a dummy string) to the location (a directory or file) specified by the pathname in *target*.

Appropriate privilege (Linux: the **CAP_SYS_ADMIN** capability) is required to mount filesystems.

Values for the *filesystemtype* argument supported by the kernel are listed in */proc/filesystems* (e.g., "btrfs", "ext4", "jfs", "xfs", "vfat", "fuse", "tmpfs", "cgroup", "proc", "mqueue", "nfs", "cifs", "iso9660"). Further types may become available when the appropriate modules are loaded.

The *data* argument is interpreted by the different filesystems. Typically it is a string of comma-separated options understood by this filesystem. See mount(8) for details of the options available for each filesystem type. This argument may be specified as NULL, if there are no options.

A call to **mount**() performs one of a number of general types of operation, depending on the bits specified in *mountflags*. The choice of which operation to perform is determined by testing the bits set in *mountflags*, with the tests being conducted in the order listed here:

* Remount an existing mount: *mountflags* includes **MS_REMOUNT**.

* Create a bind mount: *mountflags* includes **MS_BIND**.

* Move an existing mount to a new location: *mountflags* includes **MS_MOVE**.

* Create a new mount: *mountflags* includes none of the above flags.

Each of these operations is detailed later in this page. Further flags may be specified in *mountflags* to modify the behavior of **mount**(), as described below.

Additional mount flags

The list below describes the additional flags that can be specified in *mountflags*. Note that some operation types ignore some or all of these flags, as described later in this page.

MS_DIRSYNC (since Linux 2.5.19)
: Make directory changes on this filesystem synchronous. (This property can be obtained for individual directories or subtrees using chattr(1).)

MS_LAZYTIME (since Linux 4.0)
: Reduce on-disk updates of inode timestamps (atime, mtime, ctime) by maintaining these changes only in memory. The on-disk timestamps are updated only when:

(a) the inode needs to be updated for some change unrelated to file timestamps;

(b) the application employs fsync(2), syncfs(2), or sync(2);

(c) an undeleted inode is evicted from memory; or

(d) more than 24 hours have passed since the inode was written to disk.

This mount option significantly reduces writes needed to update the inode's timestamps, especially mtime and atime. However, in the event of a system crash, the atime and mtime fields on disk might be out of date by up to 24 hours.

Examples of workloads where this option could be of significant benefit include frequent random writes to preallocated files, as well as cases where the **MS_STRICTATIME** mount option is also enabled. (The advantage of combining **MS_STRICTATIME** and **MS_LAZYTIME** is that stat(2) will return the correctly updated atime, but the atime updates will be flushed to disk only in the cases listed above.)

MS_MANDLOCK
: Permit mandatory locking on files in this filesystem. (Mandatory locking must still be enabled on a per-file basis, as described in fcntl(2).) Since Linux 4.5, this mount option requires the **CAP_SYS_ADMIN** capability and a kernel configured with the **CONFIG_MANDATORY_FILE_LOCKING** option.

MS_NOATIME
: Do not update access times for (all types of) files on

this filesystem.

MS_NODEV
Do not allow access to devices (special files) on this filesystem.

MS_NODIRATIME
Do not update access times for directories on this filesystem. This flag provides a subset of the functionality provided by **MS_NOATIME**; that is, **MS_NOATIME** implies **MS_NODIRATIME**.

MS_NOEXEC
Do not allow programs to be executed from this filesystem.

MS_NOSUID
Do not honor set-user-ID and set-group-ID bits or file capabilities when executing programs from this filesystem. In addition, SELinux domain transitions require the permission *nosuid_transition*, which in turn needs also the policy capability *nnp_nosuid_transition*.

MS_RDONLY
Mount filesystem read-only.

MS_REC (since Linux 2.4.11)
Used in conjunction with **MS_BIND** to create a recursive bind mount, and in conjunction with the propagation type flags to recursively change the propagation type of all of the mounts in a subtree. See below for further details.

MS_RELATIME (since Linux 2.6.20)
When a file on this filesystem is accessed, update the file's last access time (atime) only if the current value of atime is less than or equal to the file's last modification time (mtime) or last status change time (ctime). This option is useful for programs, such as **mutt**(1), that need to know when a file has been read since it was last modified. Since Linux 2.6.30, the kernel defaults to the behavior provided by this flag (unless **MS_NOATIME** was specified), and the **MS_STRICTATIME** flag is required to obtain traditional semantics. In addition, since Linux 2.6.30, the file's last access time is always updated if it is more than 1 day old.

MS_SILENT (since Linux 2.6.17)
Suppress the display of certain (*printk*()) warning messages in the kernel log. This flag supersedes the misnamed and obsolete **MS_VERBOSE** flag (available since Linux 2.4.12), which has the same meaning.

MS_STRICTATIME (since Linux 2.6.30)
Always update the last access time (atime) when files on this filesystem are accessed. (This was the default behavior before Linux 2.6.30.) Specifying this flag overrides the effect of setting the **MS_NOATIME** and **MS_RELATIME** flags.

MS_SYNCHRONOUS
Make writes on this filesystem synchronous (as though the **O_SYNC** flag to open(2) was specified for all file opens to this filesystem).

MS_NOSYMFOLLOW (since Linux 5.10)
Do not follow symbolic links when resolving paths. Symbolic links can still be created, and readlink(1), readlink(2), realpath(1), and realpath(3) all still work properly.

From Linux 2.4 onward, some of the above flags are settable on a per-mount basis, while others apply to the superblock of the mounted filesystem, meaning that all mounts of the same filesystem share those flags. (Previously, all of the flags were per-superblock.)

The per-mount-point flags are as follows:

* Since Linux 2.4: **MS_NODEV**, **MS_NOEXEC**, and **MS_NOSUID** flags are settable on a per-mount-point basis.

* Additionally, since Linux 2.6.16: **MS_NOATIME** and **MS_NODIRATIME**.

* Additionally, since Linux 2.6.20: **MS_RELATIME**.

The following flags are per-superblock: **MS_DIRSYNC**, **MS_LAZYTIME**, **MS_MANDLOCK**, **MS_SILENT**, and **MS_SYNCHRONOUS**. The initial settings of these flags are determined on the first mount of the filesystem, and will be shared by all subsequent mounts of the same filesystem. Subsequently, the settings of the flags can be changed via a remount operation (see below). Such changes will be visible via all mounts associated with the filesystem.

Since Linux 2.6.16, **MS_RDONLY** can be set or cleared on a per-mount-point basis as well as on the underlying filesystem superblock. The mounted filesystem will be writable only if neither the filesystem nor the mountpoint are flagged as read-only.

Remounting an existing mount

An existing mount may be remounted by specifying **MS_REMOUNT** in *mountflags*. This allows you to change the *mountflags* and *data* of an existing mount without having to unmount and remount the filesystem. *target* should be the same value specified in the initial **mount**() call.

The *source* and *filesystemtype* arguments are ignored.

The *mountflags* and *data* arguments should match the values used in the original **mount**() call, except for those parameters that are being deliberately changed.

The following *mountflags* can be changed: **MS_LAZYTIME**, **MS_MANDLOCK**, **MS_NOATIME**, **MS_NODEV**, **MS_NODIRATIME**, **MS_NOEXEC**, **MS_NOSUID**, **MS_RELATIME**, **MS_RDONLY**, **MS_STRICTATIME** (whose effect is to clear the **MS_NOATIME** and **MS_RELATIME** flags), and **MS_SYNCHRONOUS**. Attempts to change the setting of the **MS_DIRSYNC** and **MS_SILENT** flags during a remount are silently ignored. Note that changes to per-superblock flags are visible via all mounts of the associated filesystem (because the per-superblock flags are shared by all mounts).

Since Linux 3.17, if none of **MS_NOATIME**, **MS_NODIRATIME**, **MS_RELATIME**, or **MS_STRICTATIME** is specified in *mountflags*, then the remount operation preserves the existing values of these flags (rather than defaulting to **MS_RELATIME**).

Since Linux 2.6.26, the **MS_REMOUNT** flag can be used with **MS_BIND** to modify only the per-mount-point flags. This is particularly useful for setting or clearing the "read-only" flag on a mount without changing the underlying filesystem. Specifying *mountflags* as:

```
    MS_REMOUNT | MS_BIND | MS_RDONLY
```

will make access through this mountpoint read-only, without affecting other mounts.

Creating a bind mount

If *mountflags* includes **MS_BIND** (available since Linux 2.4), then perform a bind mount. A bind mount makes a file or a directory subtree visible at another point within the single directory hierarchy. Bind mounts may cross filesystem boundaries and span chroot(2) jails.

The *filesystemtype* and *data* arguments are ignored.

The remaining bits (other than **MS_REC**, described below) in the *mountflags* argument are also ignored. (The bind mount has the same mount options as the underlying mount.) However, see the discussion of remounting above, for a method of making an existing bind mount read-only.

By default, when a directory is bind mounted, only that directory is mounted; if there are any submounts under the directory tree, they are not bind mounted. If the **MS_REC** flag is also specified, then a recursive bind mount operation is performed: all submounts under the *source* subtree (other than unbindable mounts) are also bind mounted at the corresponding location in the *target* subtree.

Moving a mount

If *mountflags* contains the flag **MS_MOVE** (available since Linux 2.4.18), then move a subtree: *source* specifies an existing mount and *target* specifies the new location to which that mount is to be relocated. The move is atomic: at no point is the subtree unmounted.

The remaining bits in the *mountflags* argument are ignored, as are the *filesystemtype* and *data* arguments.

Creating a new mount

If none of **MS_REMOUNT**, **MS_BIND**, **MS_MOVE**, **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or **MS_UNBINDABLE** is specified in *mountflags*, then **mount**() performs its default action: creating a new mount. *source* specifies the source for the new mount, and *target* specifies the directory at which to create the mount point.

The *filesystemtype* and *data* arguments are employed, and further bits may be specified in *mountflags* to modify the behavior of the call.

RETURN VALUE   top

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

ERRORS   top

The error values given below result from filesystem type independent errors. Each filesystem type may have its own special errors and its own special behavior. See the Linux kernel source code for details.

EACCES A component of a path was not searchable. (See also path_resolution(7).)

EACCES Mounting a read-only filesystem was attempted without giving the **MS_RDONLY** flag.

The filesystem may be read-only for various reasons, including: it resides on a read-only optical disk; it is resides on a device with a physical switch that has been set to mark the device read-only; the filesystem implementation was compiled with read-only support; or errors were detected when initially mounting the filesystem, so that it was marked read-only and can't be remounted as read-write (until the errors are fixed).

Some filesystems instead return the error **EROFS** on an attempt to mount a read-only filesystem.

EACCES The block device *source* is located on a filesystem mounted with the **MS_NODEV** option.

EBUSY An attempt was made to stack a new mount directly on top of an existing mount point that was created in this mount namespace with the same *source* and *target*.

EBUSY *source* cannot be remounted read-only, because it still holds files open for writing.

EFAULT One of the pointer arguments points outside the user address space.

EINVAL *source* had an invalid superblock.

EINVAL A remount operation (**MS_REMOUNT**) was attempted, but *source* was not already mounted on *target*.

EINVAL A move operation (**MS_MOVE**) was attempted, but the mount tree under *source* includes unbindable mounts and *target* is a mount that has propagation type **MS_SHARED**.

EINVAL A move operation (**MS_MOVE**) was attempted, but the parent mount of *source* mount has propagation type **MS_SHARED**.

EINVAL A move operation (**MS_MOVE**) was attempted, but *source* was not a mount, or was '/'.

EINVAL A bind operation (**MS_BIND**) was requested where *source* referred a mount namespace magic link (i.e., a */proc/[pid]/ns/mnt* magic link or a bind mount to such a link) and the propagation type of the parent mount of *target* was **MS_SHARED**, but propagation of the requested bind mount could lead to a circular dependency that might prevent the mount namespace from ever being freed.

EINVAL *mountflags* includes more than one of **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or **MS_UNBINDABLE**.

EINVAL *mountflags* includes **MS_SHARED**, **MS_PRIVATE**, **MS_SLAVE**, or

MS_UNBINDABLE and also includes a flag other than **MS_REC** or **MS_SILENT**.

EINVAL An attempt was made to bind mount an unbindable mount.

EINVAL In an unprivileged mount namespace (i.e., a mount namespace owned by a user namespace that was created by an unprivileged user), a bind mount operation (**MS_BIND**) was attempted without specifying (**MS_REC**), which would have revealed the filesystem tree underneath one of the submounts of the directory being bound.

ELOOP Too many links encountered during pathname resolution.

ELOOP A move operation was attempted, and *target* is a descendant of *source*.

EMFILE (In case no block device is required:) Table of dummy devices is full.

ENAMETOOLONG
A pathname was longer than **MAXPATHLEN**.

ENODEV *filesystemtype* not configured in the kernel.

ENOENT A pathname was empty or had a nonexistent component.

ENOMEM The kernel could not allocate a free page to copy filenames or data into.

ENOTBLK
source is not a block device (and a device was required).

ENOTDIR
target, or a prefix of *source*, is not a directory.

ENXIO The major number of the block device *source* is out of range.

EPERM The caller does not have the required privileges.

EPERM An attempt was made to modify (**MS_REMOUNT**) the **MS_RDONLY**, **MS_NOSUID**, or **MS_NOEXEC** flag, or one of the "atime" flags (**MS_NOATIME**, **MS_NODIRATIME**, **MS_RELATIME**) of an existing mount, but the mount is locked; see mount_namespaces(7).

EROFS Mounting a read-only filesystem was attempted without giving the **MS_RDONLY** flag. See **EACCES**, above.

VERSIONS top

The definitions of **MS_DIRSYNC**, **MS_MOVE**, **MS_PRIVATE**, **MS_REC**, **MS_RELATIME**, **MS_SHARED**, **MS_SLAVE**, **MS_STRICTATIME**, and **MS_UNBINDABLE** were added to glibc headers in version 2.12.

CONFORMING TO top

This function is Linux-specific and should not be used in programs intended to be portable.

NOTES top

Since Linux 2.4 a single filesystem can be mounted at multiple mount points, and multiple mounts can be stacked on the same mount point.

The *mountflags* argument may have the magic number 0xC0ED (**MS_MGC_VAL**) in the top 16 bits. (All of the other flags discussed in DESCRIPTION occupy the low order 16 bits of *mountflags*.) Specifying **MS_MGC_VAL** was required in kernel versions prior to 2.4, but since Linux 2.4 is no longer required and is ignored if specified.

The original **MS_SYNC** flag was renamed **MS_SYNCHRONOUS** in 1.1.69 when a different **MS_SYNC** was added to *<mman.h>*.

Before Linux 2.4 an attempt to execute a set-user-ID or set-group-ID program on a filesystem mounted with **MS_NOSUID** would fail with **EPERM**. Since Linux 2.4 the set-user-ID and set-group-ID bits are just silently ignored in this case.

Parental relationship between mounts

Each mount has a parent mount. The overall parental relationship of all mounts defines the single directory hierarchy seen by the processes within a mount namespace.

The parent of a new mount is defined when the mount is created. In the usual case, the parent of a new mount is the mount of the filesystem containing the directory or file at which the new mount is attached. In the case where a new mount is stacked on top of an existing mount, the parent of the new mount is the previous mount that was stacked at that location.

The parental relationship between mounts can be discovered via the */proc/[pid]/mountinfo* file (see below).

/proc/[pid]/mounts and /proc/[pid]/mountinfo

The Linux-specific */proc/[pid]/mounts* file exposes the list of mounts in the mount namespace of the process with the specified ID. The */proc/[pid]/mountinfo* file exposes even more information about mounts, including the propagation type and mount ID information that makes it possible to discover the parental relationship between mounts. See proc(5) and mount_namespaces(7) for details of this file.

SEE ALSO top

mountpoint(1), chroot(2), ioctl_iflags(2), **mount_settatr**(2), pivot_root(2), umount(2), mount_namespaces(7), path_resolution(7), findmnt(8), lsblk(8), mount(8), umount(8)

COLOPHON top

This page is part of release 5.13 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.