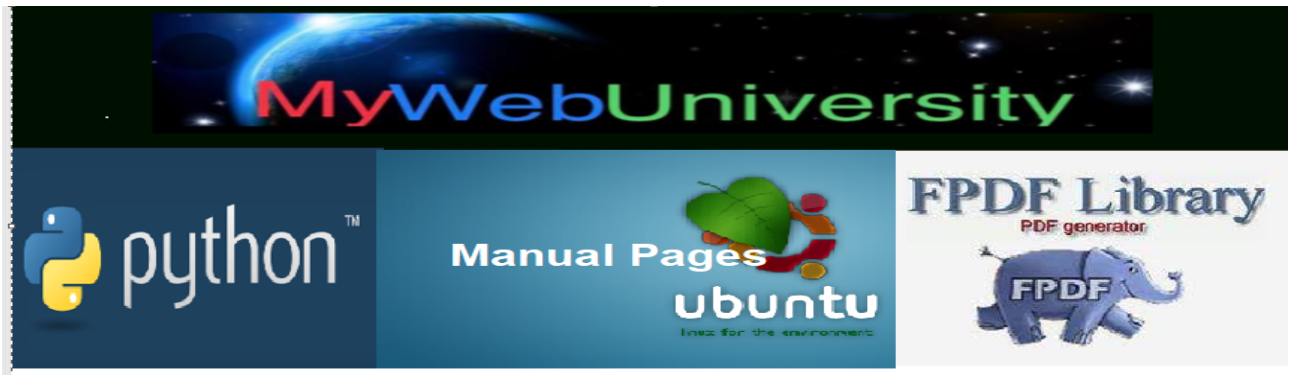

Full credit is given to the above companies including the OS that this PDF file was generated!

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

$ man mount_setattr.2

mount_setattr(2) System Calls Manual mount_setattr(2)

NAME

mount_setattr - change properties of a mount or mount tree

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <fcntl.h>       /* Definition of AT_* constants */
#include <sys/mount.h>
int mount_setattr(int dirfd, const char *path, unsigned int flags,
                  struct mount_attr *attr, size_t size);
```

DESCRIPTION

The mount_setattr() system call is part of the suite of file-descriptor-based mount facili? ties in Linux.

mount_setattr() changes the mount properties of a mount or an entire mount tree. If path is relative, then it is interpreted relative to the directory referred to by the file descrip? tor dirfd. If dirfd is the special value AT_FDCWD, then path is interpreted relative to the current working directory of the calling process. If path is the empty string and AT_EMPTY_PATH is specified in flags, then the mount properties of the mount identified by dirfd are changed. (See openat(2) for an explanation of why the dirfd argument is useful.)

The mount_setattr() system call uses an extensible structure (struct mount_attr) to allow for future extensions. Any non-flag extensions to mount_setattr() will be implemented as new fields appended to the this structure, with a zero value in a new field resulting in the kernel behaving as though that extension field was not present. Therefore, the caller must

zero-fill this structure on initialization. See the "Extensibility" subsection under NOTES for more details.

The size argument should usually be specified as sizeof(struct mount_attr). However, if the caller is using a kernel that supports an extended struct mount_attr, but the caller does not intend to make use of these features, it is possible to pass the size of an earlier ver? sion of the structure together with the extended structure. This allows the kernel to not copy later parts of the structure that aren't used anyway. With each extension that changes the size of struct mount_attr, the kernel will expose a definition of the form MOUNT_ATTR_SIZE_VERnumber. For example, the macro for the size of the initial version of struct mount_attr is MOUNT_ATTR_SIZE_VER0.

The flags argument can be used to alter the pathname resolution behavior. The supported values are:

AT_EMPTY_PATH

If path is the empty string, change the mount properties on dirfd itself.

AT_RECURSIVE

Change the mount properties of the entire mount tree.

AT_SYMLINK_NOFOLLOW

Don't follow trailing symbolic links.

AT_NO_AUTOMOUNT

Don't trigger automounts.

The attr argument of mount_setattr() is a pointer to a mount_attr structure, described in mount_attr(2type).

The attr_set and attr_clr members are used to specify the mount properties that are supposed to be set or cleared for a mount or mount tree. Flags set in attr_set enable a property on a mount or mount tree, and flags set in attr_clr remove a property from a mount or mount tree.

When changing mount properties, the kernel will first clear the flags specified in the attr_clr field, and then set the flags specified in the attr_set field. For example, these settings:

```
struct mount_attr attr = {
    .attr_clr = MOUNT_ATTR_NOEXEC | MOUNT_ATTR_NODEV,
    .attr_set = MOUNT_ATTR_RDONLY | MOUNT_ATTR_NOSUID,
};
```

are equivalent to the following steps:

```
unsigned int current_mnt_flags = mnt->mnt_flags;
/*
 * Clear all flags set in .attr_clr,
 * clearing MOUNT_ATTR_NOEXEC and MOUNT_ATTR_NODEV.
 */
current_mnt_flags &= ~attr->attr_clr;
/*
 * Now set all flags set in .attr_set,
 * applying MOUNT_ATTR_RDONLY and MOUNT_ATTR_NOSUID.
 */
current_mnt_flags |= attr->attr_set;
mnt->mnt_flags = current_mnt_flags;
```

As a result of this change, the mount or mount tree (a) is read-only; (b) blocks the execu? tion of set-user-ID and set-group-ID programs; (c) allows execution of programs; and (d) al? lows access to devices.

Multiple changes with the same set of flags requested in attr_clr and attr_set are guaran? teed to be idempotent after the changes have been applied.

The following mount attributes can be specified in the attr_set or attr_clr fields:

MOUNT_ATTR_RDONLY

If set in attr_set, makes the mount read-only. If set in attr_clr, removes the read-only setting if set on the mount.

MOUNT_ATTR_NOSUID

If set in attr_set, causes the mount not to honor the set-user-ID and set-group-ID mode bits and file capabilities when executing programs. If set in attr_clr, clears the set-user-ID, set-group-ID, and file capability restriction if set on this mount.

MOUNT_ATTR_NODEV

If set in attr_set, prevents access to devices on this mount. If set in attr_clr, removes the restriction that prevented accessing devices on this mount.

MOUNT_ATTR_NOEXEC

If set in attr_set, prevents executing programs on this mount. If set in attr_clr, removes the restriction that prevented executing programs on this mount.

MOUNT_ATTR_NOSYMFOLLOW

If set in attr_set, prevents following symbolic links on this mount. If set in attr_clr, removes the restriction that prevented following symbolic links on this mount.

MOUNT_ATTR_NODIRATIME

If set in attr_set, prevents updating access time for directories on this mount. If set in attr_clr, removes the restriction that prevented updating access time for di? rectories. Note that MOUNT_ATTR_NODIRATIME can be combined with other access-time settings and is implied by the noatime setting. All other access-time settings are mutually exclusive.

MOUNT_ATTR__ATIME - changing access-time settings

The access-time values listed below are an enumeration that includes the value zero, expressed in the bits defined by the mask MOUNT_ATTR__ATIME. Even though these bits are an enumeration (in contrast to the other mount flags such as MOUNT_ATTR_NOEXEC), they are nonetheless passed in attr_set and attr_clr for consistency with fsmount(2), which introduced this behavior.

Note that, since the access-time values are an enumeration rather than bit values, a caller wanting to transition to a different access-time setting cannot simply specify the access-time setting in attr_set, but must also include MOUNT_ATTR__ATIME in the attr_clr field. The kernel will verify that MOUNT_ATTR__ATIME isn't partially set in attr_clr (i.e., either all bits in the MOUNT_ATTR__ATIME bit field are either set or clear), and that attr_set doesn't have any access-time bits set if MOUNT_ATTR__ATIME isn't set in attr_clr.

MOUNT_ATTR_RELATIME

When a file is accessed via this mount, 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).

To enable this access-time setting on a mount or mount tree, MOUNT_ATTR_RELA? TIME must be set in attr_set and MOUNT_ATTR__ATIME must be set in the attr_clr field.

MOUNT_ATTR_NOATIME

Do not update access times for (all types of) files on this mount.

To enable this access-time setting on a mount or mount tree, MOUNT_ATTR_NOAT? IME must be set in attr_set and MOUNT_ATTR__ATIME must be set in the attr_clr

field.

MOUNT_ATTR_STRICTATIME

Always update the last access time (atime) when files are accessed on this mount.

To enable this access-time setting on a mount or mount tree, MOUNT_ATTR_STRIC? TATIME must be set in attr_set and MOUNT_ATTR__ATIME must be set in the attr_clr field.

MOUNT_ATTR_IDMAP

If set in attr_set, creates an ID-mapped mount. The ID mapping is taken from the user namespace specified in userns_fd and attached to the mount.

Since it is not supported to change the ID mapping of a mount after it has been ID mapped, it is invalid to specify MOUNT_ATTR_IDMAP in attr_clr.

For further details, see the subsection "ID-mapped mounts" under NOTES.

The propagation field is used to specify the propagation type of the mount or mount tree. This field either has the value zero, meaning leave the propagation type unchanged, or it has one of the following values:

MS_PRIVATE

Turn all mounts into private mounts.

MS_SHARED

Turn all mounts into shared mounts.

MS_SLAVE

Turn all mounts into dependent mounts.

MS_UNBINDABLE

Turn all mounts into unbindable mounts.

For further details on the above propagation types, see mount_namespaces(7).

RETURN VALUE

On success, mount_setattr() returns zero. On error, -1 is returned and errno is set to in? dicate the error.

ERRORS

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

EBADF userns_fd is not a valid file descriptor.

EBUSY The caller tried to change the mount to MOUNT_ATTR_RDONLY, but the mount still holds files open for writing.

EBUSY The caller tried to create an ID-mapped mount raising MOUNT_ATTR_IDMAP and specifying userns_fd but the mount still holds files open for writing.

EINVAL The pathname specified via the dirfd and path arguments to mount_setattr() isn't a mount point.

EINVAL An unsupported value was set in flags.

EINVAL An unsupported value was specified in the attr_set field of mount_attr.

EINVAL An unsupported value was specified in the attr_clr field of mount_attr.

EINVAL An unsupported value was specified in the propagation field of mount_attr.

EINVAL More than one of MS_SHARED, MS_SLAVE, MS_PRIVATE, or MS_UNBINDABLE was set in the propagation field of mount_attr.

EINVAL An access-time setting was specified in the attr_set field without MOUNT_ATTR__ATIME being set in the attr_clr field.

EINVAL MOUNT_ATTR_IDMAP was specified in attr_clr.

EINVAL A file descriptor value was specified in userns_fd which exceeds INT_MAX.

EINVAL A valid file descriptor value was specified in userns_fd, but the file descriptor did not refer to a user namespace.

EINVAL The underlying filesystem does not support ID-mapped mounts.

EINVAL The mount that is to be ID mapped is not a detached mount; that is, the mount has not previously been visible in a mount namespace.

EINVAL A partial access-time setting was specified in attr_clr instead of MOUNT_ATTR__ATIME being set.

EINVAL The mount is located outside the caller's mount namespace.

EINVAL The underlying filesystem has been mounted in a mount namespace that is owned by a noninitial user namespace

ENOENT A pathname was empty or had a nonexistent component.

ENOMEM When changing mount propagation to MS_SHARED, a new peer group ID needs to be allo? cated for all mounts without a peer group ID set. This allocation failed because there was not enough memory to allocate the relevant internal structures.

ENOSPC When changing mount propagation to MS_SHARED, a new peer group ID needs to be allo? cated for all mounts without a peer group ID set. This allocation failed because the kernel has run out of IDs.

EPERM One of the mounts had at least one of MOUNT_ATTR_NOATIME, MOUNT_ATTR_NODEV, MOUNT_ATTR_NODIRATIME, MOUNT_ATTR_NOEXEC, MOUNT_ATTR_NOSUID, or MOUNT_ATTR_RDONLY

set

and the flag is locked. Mount attributes become locked on a mount if:

? A new mount or mount tree is created causing mount propagation across user name? spaces (i.e., propagation to a mount namespace owned by a different user name? space). The kernel will lock the aforementioned flags to prevent these sensitive properties from being altered.

? A new mount and user namespace pair is created. This happens for example when specifying CLONE_NEWUSER | CLONE_NEWNS in unshare(2), clone(2), or clone3(2). The aforementioned flags become locked in the new mount namespace to prevent sensitive mount properties from being altered. Since the newly created mount namespace will be owned by the newly created user namespace, a calling process that is privileged in the new user namespace would?in the absence of such locking?be able to alter sensitive mount properties (e.g., to remount a mount that was marked read-only as read-write in the new mount namespace).

EPERM A valid file descriptor value was specified in userns_fd, but the file descriptor refers to the initial user namespace.

EPERM An attempt was made to add an ID mapping to a mount that is already ID mapped.

EPERM The caller does not have CAP_SYS_ADMIN in the initial user namespace.

STANDARDS

Linux.

HISTORY

Linux 5.12, glibc 2.36.

NOTES

ID-mapped mounts

Creating an ID-mapped mount makes it possible to change the ownership of all files located under a mount. Thus, ID-mapped mounts make it possible to change ownership in a temporary and localized way. It is a localized change because the ownership changes are visible only via a specific mount. All other users and locations where the filesystem is exposed are un? affected. It is a temporary change because the ownership changes are tied to the lifetime of the mount.

Whenever callers interact with the filesystem through an ID-mapped mount, the ID mapping of the mount will be applied to user and group IDs associated with filesystem objects. This encompasses the user and group IDs associated with inodes and also the following xattr(7)

keys:

? security.capability, whenever filesystem capabilities are stored or returned in the VFS_CAP_REVISION_3 format, which stores a root user ID alongside the capabilities (see capabilities(7)).

? system.posix_acl_access and system.posix_acl_default, whenever user IDs or group IDs are stored in ACL_USER or ACL_GROUP entries.

The following conditions must be met in order to create an ID-mapped mount:

? The caller must have the CAP_SYS_ADMIN capability in the user namespace the filesystem was mounted in.

? The underlying filesystem must support ID-mapped mounts. Currently, the following filesystems support ID-mapped mounts:

  ? xfs(5) (since Linux 5.12)

  ? ext4(5) (since Linux 5.12)

  ? FAT (since Linux 5.12)

  ? btrfs(5) (since Linux 5.15)

  ? ntfs3 (since Linux 5.15)

  ? f2fs (since Linux 5.18)

  ? erofs (since Linux 5.19)

  ? overlayfs (ID-mapped lower and upper layers supported since Linux 5.19)

  ? squashfs (since Linux 6.2)

  ? tmpfs (since Linux 6.3)

  ? cephfs (since Linux 6.7)

  ? hugetlbfs (since Linux 6.9)

? The mount must not already be ID-mapped. This also implies that the ID mapping of a mount cannot be altered.

? The mount must not have any writers.

? The mount must be a detached mount; that is, it must have been created by calling open_tree(2) with the OPEN_TREE_CLONE flag and it must not already have been visible in a mount namespace. (To put things another way: the mount must not have been attached to the filesystem hierarchy with a system call such as move_mount(2).)

ID mappings can be created for user IDs, group IDs, and project IDs. An ID mapping is es? sentially a mapping of a range of user or group IDs into another or the same range of user or group IDs. ID mappings are written to map files as three numbers separated by white

space. The first two numbers specify the starting user or group ID in each of the two user namespaces. The third number specifies the range of the ID mapping. For example, a mapping for user IDs such as "1000 1001 1" would indicate that user ID 1000 in the caller's user namespace is mapped to user ID 1001 in its ancestor user namespace. Since the map range is 1, only user ID 1000 is mapped.

It is possible to specify up to 340 ID mappings for each ID mapping type. If any user IDs or group IDs are not mapped, all files owned by that unmapped user or group ID will appear as being owned by the overflow user ID or overflow group ID respectively.

Further details on setting up ID mappings can be found in user_namespaces(7).

In the common case, the user namespace passed in userns_fd (together with MOUNT_ATTR_IDMAP in attr_set) to create an ID-mapped mount will be the user namespace of a container. In other scenarios it will be a dedicated user namespace associated with a user's login session as is the case for portable home directories in systemd-homed.service(8)). It is also per? fectly fine to create a dedicated user namespace for the sake of ID mapping a mount.

ID-mapped mounts can be useful in the following and a variety of other scenarios:

? Sharing files or filesystems between multiple users or multiple machines, especially in complex scenarios. For example, ID-mapped mounts are used to implement portable home di? rectories in systemd-homed.service(8), where they allow users to move their home direc? tory to an external storage device and use it on multiple computers where they are as? signed different user IDs and group IDs. This effectively makes it possible to assign random user IDs and group IDs at login time.

? Sharing files or filesystems from the host with unprivileged containers. This allows a user to avoid having to change ownership permanently through chown(2).

? ID mapping a container's root filesystem. Users don't need to change ownership perma? nently through chown(2). Especially for large root filesystems, using chown(2) can be prohibitively expensive.

? Sharing files or filesystems between containers with non-overlapping ID mappings.

? Implementing discretionary access (DAC) permission checking for filesystems lacking a concept of ownership.

? Efficiently changing ownership on a per-mount basis. In contrast to chown(2), changing ownership of large sets of files is instantaneous with ID-mapped mounts. This is espe? cially useful when ownership of an entire root filesystem of a virtual machine or con? tainer is to be changed as mentioned above. With ID-mapped mounts, a single mount_se?

tattr() system call will be sufficient to change the ownership of all files.

? Taking the current ownership into account. ID mappings specify precisely what a user or group ID is supposed to be mapped to. This contrasts with the chown(2) system call which cannot by itself take the current ownership of the files it changes into account. It simply changes the ownership to the specified user ID and group ID.

? Locally and temporarily restricted ownership changes. ID-mapped mounts make it possible to change ownership locally, restricting the ownership changes to specific mounts, and temporarily as the ownership changes only apply as long as the mount exists. By con? trast, changing ownership via the chown(2) system call changes the ownership globally and permanently.

Mount attributes and filesystem parameters

Some mount attributes (traditionally associated with mount(8)-style options) have a sibling filesystem parameter with superficially similar user-facing behavior. For example, the -o ro option to mount(8) can refer to the "read-only" filesystem parameter, or the "read-only" mount attribute. Both of these result in mount objects becoming read-only, but they do have different behavior.

The distinction between these two kinds of option is that mount object attributes are ap? plied per-mount-object (allowing different mount objects derived from a given filesystem in? stance to have different attributes), while filesystem instance parameters ("superblock flags" in kernel-developer parlance) apply to all mount objects derived from the same filesystem instance.

When using mount(2), the line between these two types of mount options was blurred. How? ever, with mount_setattr() and fsconfig(2), the distinction is made much clearer. Mount at? tributes are configured with mount_setattr(), while filesystem parameters are configured us? ing fsconfig(2).

Extensibility

In order to allow for future extensibility, mount_setattr() requires the user-space applica? tion to specify the size of the mount_attr structure that it is passing. By providing this information, it is possible for mount_setattr() to provide both forwards- and backwards-com? patibility, with size acting as an implicit version number. (Because new extension fields will always be appended, the structure size will always increase.) This extensibility de? sign is very similar to other system calls such as perf_setattr(2), perf_event_open(2), clone3(2) and openat2(2).

Let usize be the size of the structure as specified by the user-space application, and let ksize be the size of the structure which the kernel supports, then there are three cases to consider:

? If ksize equals usize, then there is no version mismatch and attr can be used verbatim.

? If ksize is larger than usize, then there are some extension fields that the kernel sup? ports which the user-space application is unaware of. Because a zero value in any added extension field signifies a no-op, the kernel treats all of the extension fields not pro? vided by the user-space application as having zero values. This provides backwards-com? patibility.

? If ksize is smaller than usize, then there are some extension fields which the user-space application is aware of but which the kernel does not support. Because any extension field must have its zero values signify a no-op, the kernel can safely ignore the unsup? ported extension fields if they are all zero. If any unsupported extension fields are non-zero, then -1 is returned and errno is set to E2BIG. This provides forwards-compati? bility.

Because the definition of struct mount_attr may change in the future (with new fields being added when system headers are updated), user-space applications should zero-fill struct mount_attr to ensure that recompiling the program with new headers will not result in spuri? ous errors at run time. The simplest way is to use a designated initializer:

```
struct mount_attr attr = {
    .attr_set = MOUNT_ATTR_RDONLY,
    .attr_clr = MOUNT_ATTR_NODEV
};
```

Alternatively, the structure can be zero-filled using memset(3) or similar functions:

```
struct mount_attr attr;
memset(&attr, 0, sizeof(attr));
attr.attr_set = MOUNT_ATTR_RDONLY;
attr.attr_clr = MOUNT_ATTR_NODEV;
```

A user-space application that wishes to determine which extensions the running kernel sup? ports can do so by conducting a binary search on size with a structure which has every byte nonzero (to find the largest value which doesn't produce an error of E2BIG).

EXAMPLES

/*

```
 * This program allows the caller to create a new detached mount
 * and set various properties on it.
 */
#define _GNU_SOURCE
#include <err.h>
#include <fcntl.h>
#include <getopt.h>
#include <sys/mount.h>
#include <sys/types.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
static const struct option longopts[] = {
    {"map-mount",       required_argument,  NULL,  'a'},
    {"recursive",       no_argument,        NULL,  'b'},
    {"read-only",       no_argument,        NULL,  'c'},
    {"block-setid",     no_argument,        NULL,  'd'},
    {"block-devices",   no_argument,        NULL,  'e'},
    {"block-exec",      no_argument,        NULL,  'f'},
    {"no-access-time",  no_argument,        NULL,  'g'},
    { NULL,             0,                  NULL,   0 },
};
int
main(int argc, char *argv[])
{
    int             fd_userns = -1;
    int             fd_tree;
    int             index = 0;
    int             ret;
    bool              recursive = false;
    const char        *source;
```

```
const char         *target;
struct mount_attr  *attr = &(struct mount_attr){};
while ((ret = getopt_long_only(argc, argv, "",
                               longopts, &index)) != -1) {
    switch (ret) {
    case 'a':
        fd_userns = open(optarg, O_RDONLY | O_CLOEXEC);
        if (fd_userns == -1)
            err(EXIT_FAILURE, "open(%s)", optarg);
        break;
    case 'b':
        recursive = true;
        break;
    case 'c':
        attr->attr_set |= MOUNT_ATTR_RDONLY;
        break;
    case 'd':
        attr->attr_set |= MOUNT_ATTR_NOSUID;
        break;
    case 'e':
        attr->attr_set |= MOUNT_ATTR_NODEV;
        break;
    case 'f':
        attr->attr_set |= MOUNT_ATTR_NOEXEC;
        break;
    case 'g':
        attr->attr_set |= MOUNT_ATTR_NOATIME;
        attr->attr_clr |= MOUNT_ATTR__ATIME;
        break;
    default:
        errx(EXIT_FAILURE, "Invalid argument specified");
    }
}
```

```
    if ((argc - optind) < 2)
        errx(EXIT_FAILURE, "Missing source or target mount point");
    source = argv[optind];
    target = argv[optind + 1];
    /* In the following, -1 as the 'dirfd' argument ensures that
       open_tree() fails if 'source' is not an absolute pathname.  */
    fd_tree = open_tree(-1, source,
                        OPEN_TREE_CLONE | OPEN_TREE_CLOEXEC |
                        AT_EMPTY_PATH | (recursive ? AT_RECURSIVE : 0));
    if (fd_tree == -1)
        err(EXIT_FAILURE, "open(%s)", source);
    if (fd_userns >= 0) {
        attr->attr_set  |= MOUNT_ATTR_IDMAP;
        attr->userns_fd = fd_userns;
    }
    ret = mount_setattr(fd_tree, "",
                        AT_EMPTY_PATH | (recursive ? AT_RECURSIVE : 0),
                        attr, sizeof(struct mount_attr));
    if (ret == -1)
        err(EXIT_FAILURE, "mount_setattr");
    close(fd_userns);
    /* In the following, -1 as the 'to_dirfd' argument ensures that
       open_tree() fails if 'target' is not an absolute pathname.  */
    ret = move_mount(fd_tree, "", -1, target,
                     MOVE_MOUNT_F_EMPTY_PATH);
    if (ret == -1)
        err(EXIT_FAILURE, "move_mount() to %s", target);
    close(fd_tree);
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

newgidmap(1), newuidmap(1), clone(2), mount(2), unshare(2), proc(5), capabilities(7), mount_namespaces(7), user_namespaces(7), xattr(7)