Rocky Enterprise Linux 9.2 Manual Pages on command 'mount_namespaces.7'

$ man mount_namespaces.7

MOUNT_NAMESPACES(7)   Linux Programmer's Manual   MOUNT_NAMESPACES(7)

NAME

   mount_namespaces - overview of Linux mount namespaces

DESCRIPTION

   For an overview of namespaces, see namespaces(7).

   Mount namespaces provide isolation of the list of mount points seen by the processes in

   each namespace instance. Thus, the processes in each of the mount namespace instances

   will see distinct single-directory hierarchies.

   The   views   provided   by   the   /proc/[pid]/mounts, /proc/[pid]/mountinfo, and

   /proc/[pid]/mountstats files (all described in proc(5)) correspond to the mount namespace

   in which the process with the PID [pid] resides. (All of the processes that reside in the

   same mount namespace will see the same view in these files.)

   A new mount namespace is created using either clone(2) or unshare(2) with the CLONE_NEWNS

   flag.   When a new mount namespace is created, its mount point list is initialized as fol?

   lows:

   * If the namespace is created using clone(2), the mount point list of the child's name?

      space is a copy of the mount point list in the parent's namespace.

   * If the namespace is created using unshare(2), the mount point list of the new namespace

      is a copy of the mount point list in the caller's previous mount namespace.

   Subsequent modifications to the mount point list (mount(2) and umount(2)) in either mount

   namespace will not (by default) affect the mount point list seen in the other namespace

   (but see the following discussion of shared subtrees).

   Restrictions on mount namespaces

Note the following points with respect to mount namespaces:

* Each mount namespace has an owner user namespace. As explained above, when a new mount namespace is created, its mount point list is initialized as a copy of the mount point list of another mount namespace. If the new namespace and the namespace from which the mount point list was copied are owned by different user namespaces, then the new mount namespace is considered less privileged.

* When creating a less privileged mount namespace, shared mounts are reduced to slave mounts. (Shared and slave mounts are discussed below.) This ensures that mappings performed in less privileged mount namespaces will not propagate to more privileged mount namespaces.

* Mounts that come as a single unit from a more privileged mount namespace are locked to‐ gether and may not be separated in a less privileged mount namespace. (The unshare(2) CLONE_NEWNS operation brings across all of the mounts from the original mount namespace as a single unit, and recursive mounts that propagate between mount namespaces propa‐ gate as a single unit.)

* The mount(2) flags MS_RDONLY, MS_NOSUID, MS_NOEXEC, and the "atime" flags (MS_NOATIME, MS_NODIRATIME, MS_RELATIME) settings become locked when propagated from a more privi‐ leged to a less privileged mount namespace, and may not be changed in the less privi‐ leged mount namespace.

* A file or directory that is a mount point in one namespace that is not a mount point in another namespace, may be renamed, unlinked, or removed (rmdir(2)) in the mount name‐ space in which it is not a mount point (subject to the usual permission checks).   Con‐ sequently, the mount point is removed in the mount namespace where it was a mount point.
  Previously (before Linux 3.18), attempting to unlink, rename, or remove a file or di‐ rectory that was a mount point in another mount namespace would result in the error EBUSY. That behavior had technical problems of enforcement (e.g., for NFS) and permit‐ ted denial-of-service attacks against more privileged users. (i.e., preventing indi‐ vidual files from being updated by bind mounting on top of them).

SHARED SUBTREES

After the implementation of mount namespaces was completed, experience showed that the isolation that they provided was, in some cases, too great. For example, in order to make a newly loaded optical disk available in all mount namespaces, a mount operation was re‐

quired in each namespace. For this use case, and others, the shared subtree feature was introduced in Linux 2.6.15. This feature allows for automatic, controlled propagation of mount and unmount events between namespaces (or, more precisely, between the members of a peer group that are propagating events to one another).

Each mount point is marked (via mount(2)) as having one of the following propagation types:

MS_SHARED

This mount point shares events with members of a peer group. Mount and unmount events immediately under this mount point will propagate to the other mount points that are members of the peer group. Propagation here means that the same mount or unmount will automatically occur under all of the other mount points in the peer group. Conversely, mount and unmount events that take place under peer mount points will propagate to this mount point.

MS_PRIVATE

This mount point is private; it does not have a peer group. Mount and unmount events do not propagate into or out of this mount point.

MS_SLAVE

Mount and unmount events propagate into this mount point from a (master) shared peer group. Mount and unmount events under this mount point do not propagate to any peer.

Note that a mount point can be the slave of another peer group while at the same time sharing mount and unmount events with a peer group of which it is a member. (More precisely, one peer group can be the slave of another peer group.)

MS_UNBINDABLE

This is like a private mount, and in addition this mount can't be bind mounted. Attempts to bind mount this mount (mount(2) with the MS_BIND flag) will fail.

When a recursive bind mount (mount(2) with the MS_BIND and MS_REC flags) is per?formed on a directory subtree, any bind mounts within the subtree are automatically pruned (i.e., not replicated) when replicating that subtree to produce the target subtree.

For a discussion of the propagation type assigned to a new mount, see NOTES.

The propagation type is a per-mount-point setting; some mount points may be marked as shared (with each shared mount point being a member of a distinct peer group), while oth?

ers are private (or slaved or unbindable).

Note that a mount's propagation type determines whether mounts and unmounts of mount points immediately under the mount point are propagated. Thus, the propagation type does not affect propagation of events for grandchildren and further removed descendant mount points. What happens if the mount point itself is unmounted is determined by the propaga‐ tion type that is in effect for the parent of the mount point.

Members are added to a peer group when a mount point is marked as shared and either:

* the mount point is replicated during the creation of a new mount namespace; or

* a new bind mount is created from the mount point.

In both of these cases, the new mount point joins the peer group of which the existing mount point is a member.

A new peer group is also created when a child mount point is created under an existing mount point that is marked as shared. In this case, the new child mount point is also marked as shared and the resulting peer group consists of all the mount points that are replicated under the peers of parent mount.

A mount ceases to be a member of a peer group when either the mount is explicitly un‐ mounted, or when the mount is implicitly unmounted because a mount namespace is removed (because it has no more member processes).

The propagation type of the mount points in a mount namespace can be discovered via the "optional fields" exposed in /proc/[pid]/mountinfo. (See proc(5) for details of this file.) The following tags can appear in the optional fields for a record in that file:

shared:X

    This mount point is shared in peer group X. Each peer group has a unique ID that is automatically generated by the kernel, and all mount points in the same peer group will show the same ID. (These IDs are assigned starting from the value 1, and may be recycled when a peer group ceases to have any members.)

master:X

    This mount is a slave to shared peer group X.

propagate_from:X (since Linux 2.6.26)

    This mount is a slave and receives propagation from shared peer group X. This tag will always appear in conjunction with a master:X tag. Here, X is the closest dom‐ inant peer group under the process's root directory. If X is the immediate master of the mount, or if there is no dominant peer group under the same root, then only

the master:X field is present and not the propagate_from:X field. For further de?

tails, see below.

unbindable

This is an unbindable mount.

If none of the above tags is present, then this is a private mount.

MS_SHARED and MS_PRIVATE example

Suppose that on a terminal in the initial mount namespace, we mark one mount point as

shared and another as private, and then view the mounts in /proc/self/mountinfo:

```
sh1# mount --make-shared /mntS

sh1# mount --make-private /mntP

sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'

77 61 8:17 / /mntS rw,relatime shared:1

83 61 8:15 / /mntP rw,relatime
```

From the /proc/self/mountinfo output, we see that /mntS is a shared mount in peer group 1,

and that /mntP has no optional tags, indicating that it is a private mount. The first two

fields in each record in this file are the unique ID for this mount, and the mount ID of

the parent mount. We can further inspect this file to see that the parent mount point of

/mntS and /mntP is the root directory, /, which is mounted as private:

```
sh1# cat /proc/self/mountinfo | awk '$1 == 61' | sed 's/ - .*//'

61 0 8:2 / / rw,relatime
```

On a second terminal, we create a new mount namespace where we run a second shell and in?

spect the mounts:

```
$ PS1='sh2# ' sudo unshare -m --propagation unchanged sh

sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'

222 145 8:17 / /mntS rw,relatime shared:1

225 145 8:15 / /mntP rw,relatime
```

The new mount namespace received a copy of the initial mount namespace's mount points.

These new mount points maintain the same propagation types, but have unique mount IDs.

(The --propagation unchanged option prevents unshare(1) from marking all mounts as private

when creating a new mount namespace, which it does by default.)

In the second terminal, we then create submounts under each of /mntS and /mntP and inspect

the set-up:

```
sh2# mkdir /mntS/a
```

```
sh2# mount /dev/sdb6 /mntS/a
sh2# mkdir /mntP/b
sh2# mount /dev/sdb7 /mntP/b
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
222 145 8:17 / /mntS rw,relatime shared:1
225 145 8:15 / /mntP rw,relatime
178 222 8:22 / /mntS/a rw,relatime shared:2
230 225 8:23 / /mntP/b rw,relatime
```

From the above, it can be seen that /mntS/a was created as shared (inheriting this setting from its parent mount) and /mntP/b was created as a private mount.

Returning to the first terminal and inspecting the set-up, we see that the new mount cre‐ated under the shared mount point /mntS propagated to its peer mount (in the initial mount namespace), but the new mount created under the private mount point /mntP did not propa‐gate:

```
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
77 61 8:17 / /mntS rw,relatime shared:1
83 61 8:15 / /mntP rw,relatime
179 77 8:22 / /mntS/a rw,relatime shared:2
```

MS_SLAVE example

Making a mount point a slave allows it to receive propagated mount and unmount events from a master shared peer group, while preventing it from propagating events to that master. This is useful if we want to (say) receive a mount event when an optical disk is mounted in the master shared peer group (in another mount namespace), but want to prevent mount and unmount events under the slave mount from having side effects in other namespaces.

We can demonstrate the effect of slaving by first marking two mount points as shared in the initial mount namespace:

```
sh1# mount --make-shared /mntX
sh1# mount --make-shared /mntY
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
```

On a second terminal, we create a new mount namespace and inspect the mount points:

```
sh2# unshare -m --propagation unchanged sh
```

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
```

168 167 8:23 / /mntX rw,relatime shared:1

169 167 8:22 / /mntY rw,relatime shared:2

In the new mount namespace, we then mark one of the mount points as a slave:

```
sh2# mount --make-slave /mntY
```

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
```

168 167 8:23 / /mntX rw,relatime shared:1

169 167 8:22 / /mntY rw,relatime master:2

From the above output, we see that /mntY is now a slave mount that is receiving propaga‐

tion events from the shared peer group with the ID 2.

Continuing in the new namespace, we create submounts under each of /mntX and /mntY:

```
sh2# mkdir /mntX/a
```

```
sh2# mount /dev/sda3 /mntX/a
```

```
sh2# mkdir /mntY/b
```

```
sh2# mount /dev/sda5 /mntY/b
```

When we inspect the state of the mount points in the new mount namespace, we see that

/mntX/a was created as a new shared mount (inheriting the "shared" setting from its parent

mount) and /mntY/b was created as a private mount:

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
```

168 167 8:23 / /mntX rw,relatime shared:1

169 167 8:22 / /mntY rw,relatime master:2

173 168 8:3 / /mntX/a rw,relatime shared:3

175 169 8:5 / /mntY/b rw,relatime

Returning to the first terminal (in the initial mount namespace), we see that the mount

/mntX/a propagated to the peer (the shared /mntX), but the mount /mntY/b was not propa‐

gated:

```
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
```

132 83 8:23 / /mntX rw,relatime shared:1

133 83 8:22 / /mntY rw,relatime shared:2

174 132 8:3 / /mntX/a rw,relatime shared:3

Now we create a new mount point under /mntY in the first shell:

```
sh1# mkdir /mntY/c
```

```
sh1# mount /dev/sda1 /mntY/c
```

```
sh1# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
132 83 8:23 / /mntX rw,relatime shared:1
133 83 8:22 / /mntY rw,relatime shared:2
174 132 8:3 / /mntX/a rw,relatime shared:3
178 133 8:1 / /mntY/c rw,relatime shared:4
```

When we examine the mount points in the second mount namespace, we see that in this case the new mount has been propagated to the slave mount point, and that the new mount is it? self a slave mount (to peer group 4):

```
sh2# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
168 167 8:23 / /mntX rw,relatime shared:1
169 167 8:22 / /mntY rw,relatime master:2
173 168 8:3 / /mntX/a rw,relatime shared:3
175 169 8:5 / /mntY/b rw,relatime
179 169 8:1 / /mntY/c rw,relatime master:4
```

MS_UNBINDABLE example

One of the primary purposes of unbindable mounts is to avoid the "mount point explosion" problem when repeatedly performing bind mounts of a higher-level subtree at a lower-level mount point. The problem is illustrated by the following shell session.

Suppose we have a system with the following mount points:

```
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
```

Suppose furthermore that we wish to recursively bind mount the root directory under sev? eral users' home directories. We do this for the first user, and inspect the mount points:

```
# mount --rbind / /home/cecilia/
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
```

```
/dev/sdb7 on /home/cecilia/mntY
```

When we repeat this operation for the second user, we start to see the explosion problem:

```
# mount --rbind / /home/henry
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sdb7 on /home/cecilia/mntY
/dev/sda1 on /home/henry
/dev/sdb6 on /home/henry/mntX
/dev/sdb7 on /home/henry/mntY
/dev/sda1 on /home/henry/home/cecilia
/dev/sdb6 on /home/henry/home/cecilia/mntX
/dev/sdb7 on /home/henry/home/cecilia/mntY
```

Under /home/henry, we have not only recursively added the /mntX and /mntY mounts, but also the recursive mounts of those directories under /home/cecilia that were created in the previous step. Upon repeating the step for a third user, it becomes obvious that the ex‑plosion is exponential in nature:

```
# mount --rbind / /home/otto
# mount | awk '{print $1, $2, $3}'
/dev/sda1 on /
/dev/sdb6 on /mntX
/dev/sdb7 on /mntY
/dev/sda1 on /home/cecilia
/dev/sdb6 on /home/cecilia/mntX
/dev/sdb7 on /home/cecilia/mntY
/dev/sda1 on /home/henry
/dev/sdb6 on /home/henry/mntX
/dev/sdb7 on /home/henry/mntY
/dev/sda1 on /home/henry/home/cecilia
/dev/sdb6 on /home/henry/home/cecilia/mntX
```

/dev/sdb7 on /home/henry/home/cecilia/mntY

/dev/sda1 on /home/otto

/dev/sdb6 on /home/otto/mntX

/dev/sdb7 on /home/otto/mntY

/dev/sda1 on /home/otto/home/cecilia

/dev/sdb6 on /home/otto/home/cecilia/mntX

/dev/sdb7 on /home/otto/home/cecilia/mntY

/dev/sda1 on /home/otto/home/henry

/dev/sdb6 on /home/otto/home/henry/mntX

/dev/sdb7 on /home/otto/home/henry/mntY

/dev/sda1 on /home/otto/home/henry/home/cecilia

/dev/sdb6 on /home/otto/home/henry/home/cecilia/mntX

/dev/sdb7 on /home/otto/home/henry/home/cecilia/mntY

The mount explosion problem in the above scenario can be avoided by making each of the new mounts unbindable. The effect of doing this is that recursive mounts of the root direc?tory will not replicate the unbindable mounts. We make such a mount for the first user:

    # mount --rbind --make-unbindable / /home/cecilia

Before going further, we show that unbindable mounts are indeed unbindable:

    # mkdir /mntZ

    # mount --bind /home/cecilia /mntZ

    mount: wrong fs type, bad option, bad superblock on /home/cecilia,

        missing codepage or helper program, or other error

        In some cases useful info is found in syslog - try

        dmesg | tail or so.

Now we create unbindable recursive bind mounts for the other two users:

    # mount --rbind --make-unbindable / /home/henry

    # mount --rbind --make-unbindable / /home/otto

Upon examining the list of mount points, we see there has been no explosion of mount points, because the unbindable mounts were not replicated under each user's directory:

    # mount | awk '{print $1, $2, $3}'

    /dev/sda1 on /

    /dev/sdb6 on /mntX

    /dev/sdb7 on /mntY

/dev/sda1 on /home/cecilia

/dev/sdb6 on /home/cecilia/mntX

/dev/sdb7 on /home/cecilia/mntY

/dev/sda1 on /home/henry

/dev/sdb6 on /home/henry/mntX

/dev/sdb7 on /home/henry/mntY

/dev/sda1 on /home/otto

/dev/sdb6 on /home/otto/mntX

/dev/sdb7 on /home/otto/mntY

Propagation type transitions

The following table shows the effect that applying a new propagation type (i.e., mount --make-xxxx) has on the existing propagation type of a mount point. The rows correspond to existing propagation types, and the columns are the new propagation settings. For rea? sons of space, "private" is abbreviated as "priv" and "unbindable" as "unbind".

```
          make-shared   make-slave     make-priv  make-unbind
?????????????????????????????????????????????????????????????????????
shared      ?shared        slave/priv [1]  priv      unbind
slave       ?slave+shared  slave [2]       priv      unbind
slave+shared ?slave+shared slave           priv      unbind
private     ?shared        priv [2]        priv      unbind
unbindable  ?shared        unbind [2]      priv      unbind
```

Note the following details to the table:

[1] If a shared mount is the only mount in its peer group, making it a slave automatically makes it private.

[2] Slaving a nonshared mount has no effect on the mount.

Bind (MS_BIND) semantics

Suppose that the following command is performed:

mount --bind A/a B/b

Here, A is the source mount point, B is the destination mount point, a is a subdirectory path under the mount point A, and b is a subdirectory path under the mount point B. The propagation type of the resulting mount, B/b, depends on the propagation types of the mount points A and B, and is summarized in the following table.

source(A)

```
             shared  private  slave      unbind
?????????????????????????????????????????????????????????????
dest(B) shared  ?shared  shared    slave+shared  invalid
        nonshared?shared  private   slave         invalid
```

Note that a recursive bind of a subtree follows the same semantics as for a bind operation on each mount in the subtree. (Unbindable mounts are automatically pruned at the target mount point.)

For further details, see Documentation/filesystems/sharedsubtree.txt in the kernel source tree.

Move (MS_MOVE) semantics

Suppose that the following command is performed:

    mount --move A B/b

Here, A is the source mount point, B is the destination mount point, and b is a subdirec‐tory path under the mount point B. The propagation type of the resulting mount, B/b, de‐pends on the propagation types of the mount points A and B, and is summarized in the fol‐lowing table.

```
                      source(A)
             shared  private  slave      unbind
?????????????????????????????????????????????????????????????
dest(B) shared  ?shared  shared    slave+shared  invalid
        nonshared?shared  private   slave         unbindable
```

Note: moving a mount that resides under a shared mount is invalid.

For further details, see Documentation/filesystems/sharedsubtree.txt in the kernel source tree.

Mount semantics

Suppose that we use the following command to create a mount point:

    mount device B/b

Here, B is the destination mount point, and b is a subdirectory path under the mount point B. The propagation type of the resulting mount, B/b, follows the same rules as for a bind mount, where the propagation type of the source mount is considered always to be private.

Unmount semantics

Suppose that we use the following command to tear down a mount point:

    unmount A

Here, A is a mount point on B/b, where B is the parent mount and b is a subdirectory path under the mount point B. If B is shared, then all most-recently-mounted mounts at b on mounts that receive propagation from mount B and do not have submounts under them are un?
mounted.

The /proc/[pid]/mountinfo propagate_from tag

The propagate_from:X tag is shown in the optional fields of a /proc/[pid]/mountinfo record in cases where a process can't see a slave's immediate master (i.e., the pathname of the master is not reachable from the filesystem root directory) and so cannot determine the chain of propagation between the mounts it can see.

In the following example, we first create a two-link master-slave chain between the mounts /mnt, /tmp/etc, and /mnt/tmp/etc. Then the chroot(1) command is used to make the /tmp/etc mount point unreachable from the root directory, creating a situation where the master of /mnt/tmp/etc is not reachable from the (new) root directory of the process.

First, we bind mount the root directory onto /mnt and then bind mount /proc at /mnt/proc so that after the later chroot(1) the proc(5) filesystem remains visible at the correct location in the chroot-ed environment.

    # mkdir -p /mnt/proc

    # mount --bind / /mnt

    # mount --bind /proc /mnt/proc

Next, we ensure that the /mnt mount is a shared mount in a new peer group (with no peers):

    # mount --make-private /mnt # Isolate from any previous peer group

    # mount --make-shared /mnt

    # cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'

    239 61 8:2 / /mnt ... shared:102

    248 239 0:4 / /mnt/proc ... shared:5

Next, we bind mount /mnt/etc onto /tmp/etc:

    # mkdir -p /tmp/etc

    # mount --bind /mnt/etc /tmp/etc

    # cat /proc/self/mountinfo | egrep '/mnt|/tmp/' | sed 's/ - .*//'

    239 61 8:2 / /mnt ... shared:102

    248 239 0:4 / /mnt/proc ... shared:5

    267 40 8:2 /etc /tmp/etc ... shared:102

Initially, these two mount points are in the same peer group, but we then make the

/tmp/etc a slave of /mnt/etc, and then make /tmp/etc shared as well, so that it can propa?

gate events to the next slave in the chain:

```
# mount --make-slave /tmp/etc
# mount --make-shared /tmp/etc
# cat /proc/self/mountinfo | egrep '/mnt|/tmp/' | sed 's/ - .*//'
239 61 8:2 / /mnt ... shared:102
248 239 0:4 / /mnt/proc ... shared:5
267 40 8:2 /etc /tmp/etc ... shared:105 master:102
```

Then we bind mount /tmp/etc onto /mnt/tmp/etc. Again, the two mount points are initially

in the same peer group, but we then make /mnt/tmp/etc a slave of /tmp/etc:

```
# mkdir -p /mnt/tmp/etc
# mount --bind /tmp/etc /mnt/tmp/etc
# mount --make-slave /mnt/tmp/etc
# cat /proc/self/mountinfo | egrep '/mnt|/tmp/' | sed 's/ - .*//'
239 61 8:2 / /mnt ... shared:102
248 239 0:4 / /mnt/proc ... shared:5
267 40 8:2 /etc /tmp/etc ... shared:105 master:102
273 239 8:2 /etc /mnt/tmp/etc ... master:105
```

From the above, we see that /mnt is the master of the slave /tmp/etc, which in turn is the

master of the slave /mnt/tmp/etc.

We then chroot(1) to the /mnt directory, which renders the mount with ID 267 unreachable

from the (new) root directory:

```
# chroot /mnt
```

When we examine the state of the mounts inside the chroot-ed environment, we see the fol?

lowing:

```
# cat /proc/self/mountinfo | sed 's/ - .*//'
239 61 8:2 / / ... shared:102
248 239 0:4 / /proc ... shared:5
273 239 8:2 /etc /tmp/etc ... master:105 propagate_from:102
```

Above, we see that the mount with ID 273 is a slave whose master is the peer group 105.

The mount point for that master is unreachable, and so a propagate_from tag is displayed,

indicating that the closest dominant peer group (i.e., the nearest reachable mount in the

slave chain) is the peer group with the ID 102 (corresponding to the /mnt mount point be?

fore the chroot(1) was performed.

VERSIONS

Mount namespaces first appeared in Linux 2.4.19.

CONFORMING TO

Namespaces are a Linux-specific feature.

NOTES

The propagation type assigned to a new mount point depends on the propagation type of the
parent mount.   If the mount point has a parent (i.e., it is a non-root mount point) and
the propagation type of the parent is MS_SHARED, then the propagation type of the new
mount is also MS_SHARED. Otherwise, the propagation type of the new mount is MS_PRIVATE.
Notwithstanding the fact that the default propagation type for new mount points is in many
cases MS_PRIVATE, MS_SHARED is typically more useful. For this reason, systemd(1) auto‐
matically remounts all mount points as MS_SHARED on system startup. Thus, on most modern
systems, the default propagation type is in practice MS_SHARED.

Since, when one uses unshare(1) to create a mount namespace, the goal is commonly to pro‐
vide full isolation of the mount points in the new namespace, unshare(1) (since util-linux
version 2.27) in turn reverses the step performed by systemd(1), by making all mount
points private in the new namespace. That is, unshare(1) performs the equivalent of the
following in the new mount namespace:

    mount --make-rprivate /

To prevent this, one can use the --propagation unchanged option to unshare(1).

An application that creates a new mount namespace directly using clone(2) or unshare(2)
may desire to prevent propagation of mount events to other mount namespaces (as is done by
unshare(1)). This can be done by changing the propagation type of mount points in the new
namespace to either MS_SLAVE or MS_PRIVATE. using a call such as the following:

    mount(NULL, "/", MS_SLAVE | MS_REC, NULL);

For a discussion of propagation types when moving mounts (MS_MOVE) and creating bind
mounts (MS_BIND), see Documentation/filesystems/sharedsubtree.txt.

EXAMPLES

See pivot_root(2).

SEE ALSO

unshare(1), clone(2), mount(2), pivot_root(2), setns(2), umount(2), unshare(2), proc(5),
namespaces(7), user_namespaces(7), findmnt(8), mount(8), pivot_root(8), umount(8)

Documentation/filesystems/sharedsubtree.txt in the kernel source tree.

COLOPHON

    This page is part of release 5.10 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/.

Linux   2020-11-01   MOUNT_NAMESPACES(7)