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 mounts 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 list is initialized as follows:

- If the namespace is created using **clone**(2), the mount list of the child's namespace is a copy of the mount list in the parent process's mount namespace.
- If the namespace is created using **unshare**(2), the mount list of the new namespace is a copy of the mount list in the caller's previous mount namespace.

Subsequent modifications to the mount list (**mount**(2) and **umount**(2)) in either mount namespace will not (by default) affect the mount list seen in the other namespace (but see the following discussion of shared subtrees).

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 required 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**(2) and **umount**(2) *events* between namespaces (or, more precisely, between the mounts that are members of a *peer group* that are propagating events to one another).

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

MS_SHARED
This mount shares events with members of a peer group. **mount**(2) and **umount**(2) e vents immediately under this mount will propagate to the other mounts that are members of the peer group. *Propagation* here means that the same **mount**(2) or **umount**(2) will automatically occur under all of the other mounts in the peer group. Conversely, **mount**(2) and **umount**(2) events that take place under peer mounts will propagate to this mount.

MS_PRIVATE
This mount is private; it does not have a peer group. **mount**(2) and **umount**(2) e vents do not propagate into or out of this mount.

MS_SLAVE
mount(2) and **umount**(2) events propagate into this mount from a (master) shared peer group. **mount**(2) and **umount**(2) events under this mount do not propagate to any peer.

Note that a mount can be the slave of another peer group while at the same time sharing **mount**(2) and **umount**(2) 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 performed 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 mounts may be marked as shared (with each shared mount being a member of a distinct peer group), while others are private (or slaved or unbindable).

Note that a mount's propagation type determines whether **mount**(2) and **umount**(2) of mounts *immediately under* the mount are propagated. Thus, the propagation type does not affect propagation of events for grandchildren and further removed descendant mounts. What happens if the mount itself is unmounted is determined by the propagation type that is in effect for the *parent* of the mount.

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

(a) the mount is replicated during the creation of a new mount namespace; or

(b) a new bind mount is created from the mount.

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

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

A mount ceases to be a member of a peer group when either the mount is explicitly unmounted, 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 mounts in a mount namespace can be discovered via the "optional fields" exposed in */proc/*pid*/mountinfo*. (See**pr oc**(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 is shared in peer group *X*. Each peer group has a unique ID that is automatically generated by the kernel, and all mounts 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 dominant 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 details, 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 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 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 inspect 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 mounts. These new mounts 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 created under the shared mount */mntS* propagated to its peer mount (in the initial mount namespace), but the new mount created under the private mount */mntP* did not propagate:

```
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 a slave allows it to receive propagated **mount**(2) and **umount**(2) 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**(2) and **umount**(2) events under the slave mount from having side effects in other namespaces.

We can demonstrate the effect of slaving by first marking two mounts 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 mounts:

```
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 mounts 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 propagation 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 mounts 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 propagated:

```
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 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 mounts in the second mount namespace, we see that in this case the new mount has been propagated to the slave mount, and that the new mount is itself 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 explosion" problem when repeatedly performing bind mounts of a higher-level subtree at a lower-level mount. The problem is illustrated by the following shell session.

Suppose we have a system with the following mounts:

```
# 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 several users' home directories. We do this for the first user, and inspect the mounts:

```
# 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 explosion 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 directory 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 mounts, we see there has been no explosion of mounts, 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. The rows correspond to existing propagation types, and the columns are the new propagation settings. For reasons 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, *B* is the destination mount, *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 mounts *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.rst* 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, *B* is the destination mount, 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 mounts *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	unbindable

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

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

Mount semantics

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

```
mount device B/b
```

Here, *B* is the destination mount, 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:

```
umount A
```

Here, *A* is a mount 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 recei ve propagation from mount *B* and do not have submounts under them are unmounted.

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**chr oot**(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 mounts 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 propagate 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 mounts 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 following:

```
# 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 before the **chroot**(1) was performed).

VERSIONS

Mount namespaces first appeared in Linux 2.4.19.

STANDARDS

Namespaces are a Linux-specific feature.

NOTES

The propagation type assigned to a new mount depends on the propagation type of the parent mount. If the mount 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 is in many cases **MS_PRIVATE**, **MS_SHARED** is typically more useful. For this reason, **systemd**(1) automatically remounts all mounts 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 provide full isolation of the mounts in the new namespace, **unshare**(1) (since *util-linux* 2.27) in turn reverses the step performed by **systemd**(1), by making all mounts private in the new namespace. That is,**unshar e**(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 mounts 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.rst*.

Restrictions on mount namespaces

Note the following points with respect to mount namespaces:

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

[2] When creating a less privileged mount namespace, shared mounts are reduced to slave mounts. This ensures that mappings performed in less privileged mount namespaces will not propagate to more privileged mount namespaces.

[3] Mounts that come as a single unit from a more privileged mount namespace are locked together 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 propagate as a single unit.)

In this context, "may not be separated" means that the mounts are locked so that they may not be individually unmounted. Consider the following example:

```
$ sudo sh
# mount --bind /dev/null /etc/shadow
# cat /etc/shadow       # Produces no output
```

The above steps, performed in a more privileged mount namespace, have created a bind mount that obscures the contents of the shadow password file, */etc/shadow*. For security reasons, it should not be possible to **umount**(2) that mount in a less privileged mount namespace, since that would reveal the contents of */etc/shadow*.

Suppose we now create a new mount namespace owned by a new user namespace. The new mount namespace will inherit copies of all of the mounts from the previous mount namespace. However, those mounts will be locked because the new mount namespace is less privileged. Consequently, an attempt to **umount**(2) the mount fails as show in the following step:

```
# unshare --user --map-root-user --mount \
              strace -o /tmp/log \
              umount /mnt/dir
umount: /etc/shadow: not mounted.
# grep '^umount' /tmp/log
umount2("/etc/shadow", 0)     = -1 EINVAL (Invalid argument)
```

The error message from **mount**(8) is a little confusing, but the **strace**(1) output reveals that the underlying **umount2**(2) system call failed with the error **EINVAL**, which is the error that the kernel returns to indicate that the mount is locked.

Note, however, that it is possible to stack (and unstack) a mount on top of one of the inherited locked mounts in a less privileged mount namespace:

```
# echo 'aaaaa' > /tmp/a    # File to mount onto /etc/shadow
# unshare --user --map-root-user --mount \
    sh -c 'mount --bind /tmp/a /etc/shadow; cat /etc/shadow'
aaaaa
# umount /etc/shadow
```

The final **umount**(8) command above, which is performed in the initial mount namespace, makes the original */etc/shadow* file once more visible in that namespace.

[4] Following on from point [3], note that it is possible to **umount**(2) an entire subtree of mounts that propagated as a unit into a less privileged mount namespace, as illustrated in the following example.

First, we create new user and mount namespaces using **unshare**(1). In the new mount namespace, the propagation type of all mounts is set to private. We then create a shared bind mount at */mnt*, and a small hierarchy of mounts underneath that mount.

```
$ PS1='ns1# ' sudo unshare --user --map-root-user \
                      --mount --propagation private bash
ns1# echo $$        # We need the PID of this shell later
778501
ns1# mount --make-shared --bind /mnt /mnt
ns1# mkdir /mnt/x
ns1# mount --make-private -t tmpfs none /mnt/x
ns1# mkdir /mnt/x/y
ns1# mount --make-private -t tmpfs none /mnt/x/y
ns1# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
986 83 8:5 /mnt /mnt rw,relatime shared:344
989 986 0:56 / /mnt/x rw,relatime
990 989 0:57 / /mnt/x/y rw,relatime
```

Continuing in the same shell session, we then create a second shell in a new user namespace and a new (less privileged) mount namespace and check the state of the propagated mounts rooted at */mnt*.

```
ns1# PS1='ns2# ' unshare --user --map-root-user \
                      --mount --propagation unchanged bash
ns2# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
1239 1204 8:5 /mnt /mnt rw,relatime master:344
```

```
1240 1239 0:56 / /mnt/x rw,relatime
1241 1240 0:57 / /mnt/x/y rw,relatime
```

Of note in the above output is that the propagation type of the mount */mnt* has been reduced to slave, as explained in point [2]. This means that submount events will propagate from the master */mnt* in "ns1", but propagation will not occur in the opposite direction.

From a separate terminal window, we then use **nsenter**(1) to enter the mount and user namespaces corresponding to "ns1". In that terminal window, we then recursively bind mount */mnt/x* at the location */mnt/ppp*.

```
$ PS1='ns3# ' sudo nsenter -t 778501 --user --mount
ns3# mount --rbind --make-private /mnt/x /mnt/ppp
ns3# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
986 83 8:5 /mnt /mnt rw,relatime shared:344
989 986 0:56 / /mnt/x rw,relatime
990 989 0:57 / /mnt/x/y rw,relatime
1242 986 0:56 / /mnt/ppp rw,relatime
1243 1242 0:57 / /mnt/ppp/y rw,relatime shared:518
```

Because the propagation type of the parent mount, */mnt*, was shared, the recursive bind mount propagated a small subtree of mounts under the slave mount */mnt* into "ns2", as can be verified by executing the following command in that shell session:

```
ns2# grep /mnt /proc/self/mountinfo | sed 's/ - .*//'
1239 1204 8:5 /mnt /mnt rw,relatime master:344
1240 1239 0:56 / /mnt/x rw,relatime
1241 1240 0:57 / /mnt/x/y rw,relatime
1244 1239 0:56 / /mnt/ppp rw,relatime
1245 1244 0:57 / /mnt/ppp/y rw,relatime master:518
```

While it is not possible to **umount**(2) a part of the propagated subtree (*/mnt/ppp/y*) in "ns2", it is possible to **umount**(2) the entire subtree, as shown by the following commands:

```
ns2# umount /mnt/ppp/y
umount: /mnt/ppp/y: not mounted.
ns2# umount -l /mnt/ppp | sed 's/ - .*//'      # Succeeds...
ns2# grep /mnt /proc/self/mountinfo
1239 1204 8:5 /mnt /mnt rw,relatime master:344
1240 1239 0:56 / /mnt/x rw,relatime
1241 1240 0:57 / /mnt/x/y rw,relatime
```

[5] 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 privileged to a less privileged mount namespace, and may not be changed in the less privileged mount namespace.

This point is illustrated in the following example where, in a more privileged mount namespace, we create a bind mount that is marked as read-only. For security reasons, it should not be possible to make the mount writable in a less privileged mount namespace, and indeed the kernel prevents this:

```
$ sudo mkdir /mnt/dir
$ sudo mount --bind -o ro /some/path /mnt/dir
$ sudo unshare --user --map-root-user --mount \
                mount -o remount,rw /mnt/dir
mount: /mnt/dir: permission denied.
```

[6] 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 namespace in which it is not a mount point (subject to the usual permission checks). Consequently, 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 directory 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 permitted denial-of-service attacks against more privileged users (i.e., preventing individual files from being updated by bind mounting on top of them).

EXAMPLES

See **pivot_root**(2).

SEE ALSO

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

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