Shared Subtrees

1) Overview

Consider the following situation:

A process wants to clone its own namespace, but still wants to access the CD that got mounted recently. Shared subtree semantics provide the necessary mechanism to accomplish the above.

It provides the necessary building blocks for features like per-user-namespace and versioned filesystem.

2) Features

Shared subtree provides four different flavors of mounts; struct vfsmount to be precise

a. shared mount
b. slave mount
c. private mount
d. unbindable mount

2a) A shared mount can be replicated to as many mountpoints and all the replicas continue to be exactly same.

Here is an example:

Let's say /mnt has a mount that is shared:

```
mount --make-shared /mnt
```

Note: mount(8) command now supports the --make-shared flag, so the sample 'smount' program is no longer needed and has been removed.

```
# mount --bind /mnt /tmp
```

The above command replicates the mount at /mnt to the mountpoint /tmp and the contents of both the mounts remain identical.

```
#ls /mnt
a b c

#ls /tmp
a b c
```

Now let's say we mount a device at /tmp/a:

```
# mount /dev/sd0  /tmp/a

#ls /tmp/a
t1 t2 t3

#ls /mnt/a
t1 t2 t3
```

Note that the mount has propagated to the mount at /mnt as well.

And the same is true even when /dev/sd0 is mounted on /mnt/a. The contents will be visible under /tmp/a too.

2b) A slave mount is like a shared mount except that mount and umount events

only propagate towards it.

All slave mounts have a master mount which is a shared.

Here is an example:

Let's say /mnt has a mount which is shared. # mount --make-shared /mnt

Let's bind mount /mnt to /tmp # mount --bind /mnt /tmp

the new mount at /tmp becomes a shared mount and it is a replica of the mount at /mnt.

Now let's make the mount at /tmp; a slave of /mnt # mount --make-slave /tmp

let's mount /dev/sd0 on /mnt/a # mount /dev/sd0 /mnt/a

#ls /mnt/a t1 t2 t3

#ls /tmp/a t1 t2 t3

Note the mount event has propagated to the mount at /tmp

However let's see what happens if we mount something on the mount at /tmp

mount /dev/sd1 /tmp/b

#ls /tmp/b s1 s2 s3

#ls /mnt/b

Note how the mount event has not propagated to the mount at /mnt

2c) A private mount does not forward or receive propagation.

This is the mount we are familiar with. Its the default type.

2d) A unbindable mount is a unbindable private mount

let's say we have a mount at /mnt and we make it unbindable:

```
# mount --make-unbindable /mnt
```

```
Let's try to bind mount this mount somewhere else::
```

```
# mount --bind /mnt /tmp
mount: wrong fs type, bad option, bad superblock on /mnt,
        or too many mounted file systems
```

Binding a unbindable mount is a invalid operation.

3. Setting mount states

The mount command (util-linux package) can be used to set mount states:

```
mount --make-shared mountpoint
mount --make-slave mountpoint
mount --make-private mountpoint
mount --make-unbindable mountpoint
```

4) Use cases

A. A process wants to clone its own namespace, but still wants to access the CD that got mounted recently.

Solution:

The system administrator can make the mount at /cdrom shared:

```
mount --bind /cdrom /cdrom
mount --make-shared /cdrom
```

Now any process that clones off a new namespace will have a mount at /cdrom which is a replica of the same mount in the parent namespace.

So when a CD is inserted and mounted at /cdrom that mount gets propagated to the other mount at /cdrom in all the other clone namespaces.

B) A process wants its mounts invisible to any other process, but still be able to see the other system mounts.

Solution:

To begin with, the administrator can mark the entire mount tree as shareable:

```
mount --make-rshared /
```

A new process can clone off a new namespace. And mark some part of its namespace as slave:

```
mount --make-rslave /myprivatetree
```

Hence forth any mounts within the /myprivatetree done by the process will not show up in any other namespace. However mounts done in the parent namespace under /myprivatetree still shows up in the process's namespace.

Apart from the above semantics this feature provides the building blocks to solve the following problems:

C. Per-user namespace

The above semantics allows a way to share mounts across namespaces. But namespaces are associated with processes. If namespaces are made first class objects with user API to associate/disassociate a namespace with userid, then each user could have his/her own namespace and tailor it to his/her requirements. This needs to be supported in PAM.

D. Versioned files

If the entire mount tree is visible at multiple locations, then an underlying versioning file system can return different versions of the file depending on the path used to access that file.

An example is:

```
mount --make-shared /
mount --rbind / /view/v1
mount --rbind / /view/v2
mount --rbind / /view/v3
mount --rbind / /view/v4
```

and if /usr has a versioning filesystem mounted, then that mount appears at /view/v1/usr, /view/v2/usr, /view/v3/usr and /view/v4/usr too

A user can request v3 version of the file /usr/fs/namespace.c by accessing /view/v3/usr/fs/namespace.c . The underlying versioning filesystem can then decipher that v3 version of the filesystem is being requested and return the corresponding inode.

5) Detailed semantics

The section below explains the detailed semantics of bind, rbind, move, mount, umount and clone-namespace operations.

Note: the word 'vfsmount' and the noun 'mount' have been used to mean the same thing, throughout this document.

5a) Mount states

A given mount can be in one of the following states

1. shared
2. slave
3. shared and slave
4. private
5. unbindable

A 'propagation event' is defined as event generated on a vfsmount that leads to mount or unmount actions in other vfsmounts.

A 'peer group' is defined as a group of vfsmounts that propagate events to each other.

1. Shared mounts

   A 'shared mount' is defined as a vfsmount that belongs to a 'peer group'.

   For example:

   ```
   mount --make-shared /mnt
   mount --bind /mnt /tmp
   ```

   The mount at /mnt and that at /tmp are both shared and belong to the same peer group. Anything mounted or unmounted under /mnt or /tmp reflect in all the other mounts of its peer group.

2. Slave mounts

   A 'slave mount' is defined as a vfsmount that receives propagation events and does not forward propagation events.

   A slave mount as the name implies has a master mount from which mount/unmount events are received. Events do not propagate from the slave mount to the master. Only a shared mount can be made a slave by executing the following command:

   ```
   mount --make-slave mount
   ```

   A shared mount that is made as a slave is no more shared unless modified to become shared.

3. Shared and Slave

   A vfsmount can be both shared as well as slave. This state indicates that the mount is a slave of some vfsmount, and has its own peer group too. This vfsmount receives propagation events from its master vfsmount, and also forwards propagation events to its 'peer group' and to its slave vfsmounts.

   Strictly speaking, the vfsmount is shared having its own peer group, and this peer-group is a slave of some other peer group.

Only a slave vfsmount can be made as 'shared and slave' by either executing the following command:

```
mount --make-shared mount
```

or by moving the slave vfsmount under a shared vfsmount.

4. Private mount

   A 'private mount' is defined as vfsmount that does not receive or forward any propagation events.

5. Unbindable mount

   A 'unbindable mount' is defined as vfsmount that does not receive or forward any propagation events and cannot be bind mounted.

State diagram:

The state diagram below explains the state transition of a mount, in response to various commands:

```
------------------------------------------------------------------------
|             |make-shared |  make-slave  | make-private |make-unbindab|
--------------|------------|--------------|--------------|-------------|
|shared       |shared      |*slave/private|   private    | unbindable  |
|             |            |              |              |             |
|-------------|------------|--------------|--------------|-------------|
|slave        |shared      | **slave      |    private   | unbindable  |
|             |and slave   |              |              |             |
|-------------|------------|--------------|--------------|-------------|
|shared       |shared      | slave        |    private   | unbindable  |
|and slave    |and slave   |              |              |             |
|-------------|------------|--------------|--------------|-------------|
|private      |shared      |  **private   |    private   | unbindable  |
|-------------|------------|--------------|--------------|-------------|
|unbindable   |shared      |**unbindable  |    private   | unbindable  |
------------------------------------------------------------------------

* if the shared mount is the only mount in its peer group, making it
slave, makes it private automatically. Note that there is no master to
which it can be slaved to.

** slaving a non-shared mount has no effect on the mount.
```

Apart from the commands listed below, the 'move' operation also changes the state of a mount depending on type of the destination mount. Its explained in section 5d.

5b) Bind semantics

Consider the following command:

```
mount --bind A/a  B/b
```

where 'A' is the source mount, 'a' is the dentry in the mount 'A', 'B' is the destination mount and 'b' is the dentry in the destination mount.

The outcome depends on the type of mount of 'A' and 'B'. The table below contains quick reference:

```
---------------------------------------------------------------------------
|         BIND MOUNT OPERATION                                            |
|**************************************************************************|
|source(A)->| shared       |       private  |       slave    | unbindable |
| dest(B)   |              |                |                |            |
|   |       |              |                |                |            |
|   v       |              |                |                |            |
|**************************************************************************|
|  shared   | shared       |     shared     | shared & slave |  invalid   |
|           |              |                |                |            |
|non-shared| shared        |      private   |      slave     |  invalid   |
***************************************************************************
```

Details:

1. 'A' is a shared mount and 'B' is a shared mount. A new mount 'C'
   which is clone of 'A', is created. Its root dentry is 'a' . 'C' is mounted on mount 'B' at dentry 'b'. Also new mount 'C1', 'C2', 'C3' ... are created and mounted at the dentry 'b' on all mounts where 'B' propagates to. A new propagation tree containing 'C1',..,'Cn' is created. This propagation tree is identical to the propagation tree of 'B'. And finally the peer-group of 'C' is merged with the peer group

of 'A'.

2. 'A' is a private mount and 'B' is a shared mount. A new mount 'C' which is clone of 'A', is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. Also new mount 'C1', 'C2', 'C3' ... are created and mounted at the dentry 'b' on all mounts where 'B' propagates to. A new propagation tree is set containing all new mounts 'C', 'C1', .., 'Cn' with exactly the same configuration as the propagation tree for 'B'.
3. 'A' is a slave mount of mount 'Z' and 'B' is a shared mount. A new mount 'C' which is clone of 'A', is created. Its root dentry is 'a' . 'C' is mounted on mount 'B' at dentry 'b'. Also new mounts 'C1', 'C2', 'C3' ... are created and mounted at the dentry 'b' on all mounts where 'B' propagates to. A new propagation tree containing the new mounts 'C','C1',.. 'Cn' is created. This propagation tree is identical to the propagation tree for 'B'. And finally the mount 'C' and its peer group is made the slave of mount 'Z'. In other words, mount 'C' is in the state 'slave and shared'.
4. 'A' is a unbindable mount and 'B' is a shared mount. This is a invalid operation.
5. 'A' is a private mount and 'B' is a non-shared(private or slave or unbindable) mount. A new mount 'C' which is clone of 'A', is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'.
6. 'A' is a shared mount and 'B' is a non-shared mount. A new mount 'C' which is a clone of 'A' is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. 'C' is made a member of the peer-group of 'A'.
7. 'A' is a slave mount of mount 'Z' and 'B' is a non-shared mount. A new mount 'C' which is a clone of 'A' is created. Its root dentry is 'a'. 'C' is mounted on mount 'B' at dentry 'b'. Also 'C' is set as a slave mount of 'Z'. In other words 'A' and 'C' are both slave mounts of 'Z'. All mount/unmount events on 'Z' propagates to 'A' and 'C'. But mount/unmount on 'A' do not propagate anywhere else. Similarly mount/unmount on 'C' do not propagate anywhere else.
8. 'A' is a unbindable mount and 'B' is a non-shared mount. This is a invalid operation. A unbindable mount cannot be bind mounted.

5c) Rbind semantics

rbind is same as bind. Bind replicates the specified mount. Rbind replicates all the mounts in the tree belonging to the specified mount. Rbind mount is bind mount applied to all the mounts in the tree.

If the source tree that is rbind has some unbindable mounts, then the subtree under the unbindable mount is pruned in the new location.

eg:

let's say we have the following mount tree:

```
    A
  /   \
  B   C
 / \ / \
 D E F G
```

Let's say all the mount except the mount C in the tree are of a type other than unbindable.

If this tree is rbound to say Z

We will have the following tree at the new location:

```
     Z
     |
     A'
    /
   B'                 Note how the tree under C is pruned
  / \                 in the new location.
 D' E'
```

5d) Move semantics

Consider the following command

mount --move A B/b

where 'A' is the source mount, 'B' is the destination mount and 'b' is the dentry in the destination mount.

The outcome depends on the type of the mount of 'A' and 'B'. The table below is a quick reference:

```
---------------------------------------------------------------------------
|                   MOVE MOUNT OPERATION                                  |
|**************************************************************************
| source(A)->| shared       |       private  |       slave    | unbindable |
| dest(B)  |              |                |                |            |
|   |      |              |                |                |            |
|   v      |              |                |                |            |
|**************************************************************************
|  shared  | shared       |     shared     |shared and slave|  invalid   |
|          |              |                |                |            |
|non-shared| shared       |      private   |    slave       | unbindable |
***************************************************************************
```

> **Note**
>
> moving a mount residing under a shared mount is invalid.

Details follow:

1. 'A' is a shared mount and 'B' is a shared mount. The mount 'A' is
   mounted on mount 'B' at dentry 'b'. Also new mounts 'A1', 'A2'...'An' are created and mounted at dentry 'b' on all mounts that receive propagation from mount 'B'. A new propagation tree is created in the exact same configuration as that of 'B'. This new propagation tree contains all the new mounts 'A1', 'A2'... 'An'. And this new propagation tree is appended to the already existing propagation tree of 'A'.
2. 'A' is a private mount and 'B' is a shared mount. The mount 'A' is
   mounted on mount 'B' at dentry 'b'. Also new mount 'A1', 'A2'... 'An' are created and mounted at dentry 'b' on all mounts that receive propagation from mount 'B'. The mount 'A' becomes a shared mount and a propagation tree is created which is identical to that of 'B'. This new propagation tree contains all the new mounts 'A1', 'A2'... 'An'.
3. 'A' is a slave mount of mount 'Z' and 'B' is a shared mount. The
   mount 'A' is mounted on mount 'B' at dentry 'b'. Also new mounts 'A1', 'A2'... 'An' are created and mounted at dentry 'b' on all mounts that receive propagation from mount 'B'. A new propagation tree is created in the exact same configuration as that of 'B'. This new propagation tree contains all the new mounts 'A1', 'A2'... 'An'. And this new propagation tree is appended to the already existing propagation tree of 'A'. Mount 'A' continues to be the slave mount of 'Z' but it also becomes 'shared'.
4. 'A' is a unbindable mount and 'B' is a shared mount. The operation
   is invalid. Because mounting anything on the shared mount 'B' can create new mounts that get mounted on the mounts that receive propagation from 'B'. And since the mount 'A' is unbindable, cloning it to mount at other mountpoints is not possible.
5. 'A' is a private mount and 'B' is a non-shared(private or slave or
   unbindable) mount. The mount 'A' is mounted on mount 'B' at dentry 'b'.
6. 'A' is a shared mount and 'B' is a non-shared mount. The mount 'A'
   is mounted on mount 'B' at dentry 'b'. Mount 'A' continues to be a shared mount.
7. 'A' is a slave mount of mount 'Z' and 'B' is a non-shared mount.
   The mount 'A' is mounted on mount 'B' at dentry 'b'. Mount 'A' continues to be a slave mount of mount 'Z'.
8. 'A' is a unbindable mount and 'B' is a non-shared mount. The mount
   'A' is mounted on mount 'B' at dentry 'b'. Mount 'A' continues to be a unbindable mount.

5e) Mount semantics

Consider the following command:

```
mount device  B/b
```

'B' is the destination mount and 'b' is the dentry in the destination mount.

The above operation is the same as bind operation with the exception that the source mount is always a private mount.

5f) Unmount semantics

Consider the following command:

```
umount A
```

where 'A' is a mount mounted on mount 'B' at dentry 'b'.

If mount 'B' is shared, then all most-recently-mounted mounts at dentry 'b' on mounts that receive propagation from mount 'B' and does not have sub-mounts within them are unmounted.

Example: Let's say 'B1', 'B2', 'B3' are shared mounts that propagate to each other.

let's say 'A1', 'A2', 'A3' are first mounted at dentry 'b' on mount 'B1', 'B2' and 'B3' respectively.

let's say 'C1', 'C2', 'C3' are next mounted at the same dentry 'b' on mount 'B1', 'B2' and 'B3' respectively.

if 'C1' is unmounted, all the mounts that are most-recently-mounted on 'B1' and on the mounts that 'B1' propagates-to are unmounted.

'B1' propagates to 'B2' and 'B3'. And the most recently mounted mount on 'B2' at dentry 'b' is 'C2', and that of mount 'B3' is 'C3'.

So all 'C1', 'C2' and 'C3' should be unmounted.

If any of 'C2' or 'C3' has some child mounts, then that mount is not unmounted, but all other mounts are unmounted. However if 'C1' is told to be unmounted and 'C1' has some sub-mounts, the umount operation is failed entirely.

5g) Clone Namespace

A cloned namespace contains all the mounts as that of the parent namespace.

Let's say 'A' and 'B' are the corresponding mounts in the parent and the child namespace.

If 'A' is shared, then 'B' is also shared and 'A' and 'B' propagate to each other.

If 'A' is a slave mount of 'Z', then 'B' is also the slave mount of 'Z'.

If 'A' is a private mount, then 'B' is a private mount too.

If 'A' is unbindable mount, then 'B' is a unbindable mount too.

6. Quiz

A. What is the result of the following command sequence?

```
mount --bind /mnt /mnt
mount --make-shared /mnt
mount --bind /mnt /tmp
mount --move /tmp /mnt/1
```

what should be the contents of /mnt /mnt/1 /mnt/1/1 should be? Should they all be identical? or should /mnt and /mnt/1 be identical only?

B. What is the result of the following command sequence?

```
mount --make-rshared /
mkdir -p /v/1
mount --rbind / /v/1
```

what should be the content of /v/1/v/1 be?

C. What is the result of the following command sequence?

```
mount --bind /mnt /mnt
mount --make-shared /mnt
mkdir -p /mnt/1/2/3 /mnt/1/test
mount --bind /mnt/1 /tmp
mount --make-slave /mnt
mount --make-shared /mnt
mount --bind /mnt/1/2 /tmp1
mount --make-slave /mnt
```

At this point we have the first mount at /tmp and its root dentry is 1. Let's call this mount 'A' And then we have a second mount at /tmp1 with root dentry 2. Let's call this mount 'B' Next we have a third mount at /mnt with root dentry mnt. Let's call this mount 'C'

'B' is the slave of 'A' and 'C' is a slave of 'B' A -> B -> C

at this point if we execute the following command

mount --bind /bin /tmp/test

The mount is attempted on 'A'

will the mount propagate to 'B' and 'C' ?

what would be the contents of /mnt/1/test be?

7. FAQ

Q1. Why is bind mount needed? How is it different from symbolic links?

symbolic links can get stale if the destination mount gets unmounted or moved. Bind mounts continue to exist even if the other mount is unmounted or moved.

Q2. Why can't the shared subtree be implemented using exportfs?

exportfs is a heavyweight way of accomplishing part of what shared subtree can do. I cannot imagine a way to implement the semantics of slave mount using exportfs?

Q3 Why is unbindable mount needed?

Let's say we want to replicate the mount tree at multiple locations within the same subtree.

if one rbind mounts a tree within the same subtree 'n' times the number of mounts created is an exponential function of 'n'. Having unbindable mount can help prune the unneeded bind mounts. Here is an example.

step 1:

let's say the root tree has just two directories with one vfsmount:

```
                root
               /    \
              tmp    usr

And we want to replicate the tree at multiple
mountpoints under /root/tmp
```

step 2:

```
mount --make-shared /root

mkdir -p /tmp/m1

mount --rbind /root /tmp/m1
```

the new tree now looks like this:

```
              root
             /    \
           tmp    usr
          /
         m1
        /  \
       tmp  usr
       /
      m1

it has two vfsmounts
```

step 3:

```
     mkdir -p /tmp/m2
     mount --rbind /root /tmp/m2

 the new tree now looks like this::

               root
              /    \
            tmp     usr
           /    \
         m1       m2
        / \       /  \
      tmp  usr   tmp  usr
      / \          /
     m1  m2      m1
          / \     /  \
        tmp usr  tmp   usr
        /        / \
       m1       m1  m2
      /  \
    tmp   usr
    /  \
   m1   m2

it has 6 vfsmounts
```

step 4:

::

mkdir -p /tmp/m3 mount --rbind /root /tmp/m3

I won't draw the tree..but it has 24 vfsmounts

at step i the number of vfsmounts is V[i] = i*V[i-1]. This is an exponential function. And this tree has way more mounts than what we really needed in the first place.

One could use a series of umount at each step to prune out the unneeded mounts. But there is a better solution. Unclonable mounts come in handy here.

step 1:

let's say the root tree has just two directories with one vfsmount:

```
            root
           /    \
          tmp    usr

How do we set up the same tree at multiple locations under
/root/tmp
```

step 2:

```
mount --bind /root/tmp /root/tmp

mount --make-rshared /root
mount --make-unbindable /root/tmp

mkdir -p /tmp/m1

mount --rbind /root /tmp/m1
```

the new tree now looks like this:

```
       root
      /    \
    tmp    usr
   /
  m1
 /  \
tmp  usr
```

step 3:

```
mkdir -p /tmp/m2
mount --rbind /root /tmp/m2
```

the new tree now looks like this:

```
       root
      /    \
    tmp    usr
   /   \
  m1     m2
 /  \     / \
tmp  usr tmp usr
```

step 4:

```
mkdir -p /tmp/m3
mount --rbind /root /tmp/m3
```

the new tree now looks like this:

```
             root
         /           \
        tmp           usr
    /    \    \
  m1     m2     m3
 /  \     / \    /  \
tmp  usr tmp usr tmp usr
```

8. Implementation

8A) Datastructure

4 new fields are introduced to struct vfsmount:

- ->mnt_share
- ->mnt_slave_list
- ->mnt_slave
- ->mnt_master

->mnt_share

links together all the mount to/from which this vfsmount send/receives propagation events.

->mnt_slave_list
: links all the mounts to which this vfsmount propagates to.

->mnt_slave
: links together all the slaves that its master vfsmount propagates to.

->mnt_master
: points to the master vfsmount from which this vfsmount receives propagation.

->mnt_flags
: takes two more flags to indicate the propagation status of the vfsmount. MNT_SHARE indicates that the vfsmount is a shared vfsmount. MNT_UNCLONABLE indicates that the vfsmount cannot be replicated.

All the shared vfsmounts in a peer group form a cyclic list through ->mnt_share.

All vfsmounts with the same ->mnt_master form on a cyclic list anchored in ->mnt_master->mnt_slave_list and going through ->mnt_slave.

> ->mnt_master can point to arbitrary (and possibly different) members of master peer group. To find all immediate slaves of a peer group you need to go through _all_ ->mnt_slave_list of its members. Conceptually it's just a single set - distribution among the individual lists does not affect propagation or the way propagation tree is modified by operations.

All vfsmounts in a peer group have the same ->mnt_master. If it is non-NULL, they form a contiguous (ordered) segment of slave list.

A example propagation tree looks as shown in the figure below. [NOTE: Though it looks like a forest, if we consider all the shared mounts as a conceptual entity called 'pnode', it becomes a tree]:

```
        A <--> B <--> C <---> D
       /|\            /|      |\
      / F G          J K      H I
     /
    E<-->K
        /|\
       M L N
```

In the above figure A,B,C and D all are shared and propagate to each other. 'A' has got 3 slave mounts 'E' 'F' and 'G' 'C' has got 2 slave mounts 'J' and 'K' and 'D' has got two slave mounts 'H' and 'I'. 'E' is also shared with 'K' and they propagate to each other. And 'K' has 3 slaves 'M', 'L' and 'N'

A's ->mnt_share links with the ->mnt_share of 'B' 'C' and 'D'

A's ->mnt_slave_list links with ->mnt_slave of 'E', 'K', 'F' and 'G'

E's ->mnt_share links with ->mnt_share of K

'E', 'K', 'F', 'G' have their ->mnt_master point to struct vfsmount of 'A'

'M', 'L', 'N' have their ->mnt_master point to struct vfsmount of 'K'

K's ->mnt_slave_list links with ->mnt_slave of 'M', 'L' and 'N'

C's ->mnt_slave_list links with ->mnt_slave of 'J' and 'K'

J and K's ->mnt_master points to struct vfsmount of C

and finally D's ->mnt_slave_list links with ->mnt_slave of 'H' and 'I'

'H' and 'I' have their ->mnt_master pointing to struct vfsmount of 'D'.

NOTE: The propagation tree is orthogonal to the mount tree.

8B Locking:

->mnt_share, ->mnt_slave, ->mnt_slave_list, ->mnt_master are protected by namespace_sem (exclusive for modifications, shared for reading).

Normally we have ->mnt_flags modifications serialized by vfsmount_lock. There are two exceptions: do_add_mount() and clone_mnt(). The former modifies a vfsmount that has not been visible in any shared data structures yet. The latter holds namespace_sem and the only references to vfsmount are in lists that can't be traversed without namespace_sem.

8C Algorithm:

The crux of the implementation resides in rbind/move operation.

The overall algorithm breaks the operation into 3 phases: (look at attach_recursive_mnt() and propagate_mnt())

1. prepare phase.
2. commit phases.

3. abort phases.

Prepare phase:

for each mount in the source tree:

a. Create the necessary number of mount trees to
   be attached to each of the mounts that receive propagation from the destination mount.
b. Do not attach any of the trees to its destination. However note down its ->mnt_parent and ->mnt_mountpoint
c. Link all the new mounts to form a propagation tree that is identical to the propagation tree of the destination mount.

If this phase is successful, there should be 'n' new propagation trees; where 'n' is the number of mounts in the source tree. Go to the commit phase

Also there should be 'm' new mount trees, where 'm' is the number of mounts to which the destination mount propagates to.

if any memory allocations fail, go to the abort phase.

Commit phase
attach each of the mount trees to their corresponding destination mounts.

Abort phase
delete all the newly created trees.

> **Note**
>
> all the propagation related functionality resides in the file pnode.c

---

version 0.1 (created the initial document, Ram Pai linuxram@us.ibm.com)

version 0.2 (Incorporated comments from Al Viro)