Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Shared-Subtree Concept, Implementation, and Applications in Linux

Al Viro
Red Hat, Inc.
viro@ftp.linux.org.uk

Ram Pai
IBM Corporation
linuxram@us.ibm.com

Abstract

Concepts like per-process namespaces and bind mounts have enriched the Linux® VFS for a couple of years now. Various solutions have attempted to use these features for customized mount setups in a virtualized environment and for setting up mirrored mount trees to support versioned filesystems.

But more than often, the isolated nature of per-process namespaces and the static nature of bind mount have restricted their use. Consequently a new VFS enhancement called shared subtree was introduced in the Linux 2.6.15 kernel. This enhancement makes per-process namespaces and bind mounts dynamic in nature and provides a crucial building block for various solutions.

In this paper we describe shared subtree semantics and their application in real life. We also discuss the design and implementation details of the feature.

1 Introduction

The Linux VFS provides a rich set of features to tailor access to files and filesystems.

The mount feature provides a convenient way to access the contents of a filesystem. Using the mount abstraction, you can also mount other filesystems over a directory of an existing filesystem, thus creating a filesystem mount tree.

Linux further allows the same filesystem to be mounted at different locations within the filesystem mount tree, thus providing multiple paths to access the same filesystem.

Adding to this set of features, you can pick a directory tree in the filesystem mount tree and mount it at some other location using the recursive bind feature. The filesystem mount tree can further be moved across locations through the move mount feature.

And finally, Linux allows a new process to fork an entirely new filesystem mount tree to which the process is associated using the filesystem namespace feature. This feature is referred to as per-process namespace.

However, features like bind and filesystem namespace have not seen many applications in real life. For example, the filesystem-namespace feature isolates a process; i.e. a process that associates with a namespace does not see new mounts in another namespace, nor does it propagate its own mounts to a different namespace.

Different projects like FUSE, SELinux's Labelled System Security Profile (LSPP) system, and Multiple Versioned File System MVFS™ have considered using the filesystem-namespace and recursive-bind features. But the static nature of these features has often restricted their use. The following four scenarios illustrate the problem.

FUSE (**F**ilesystem in **U**ser **S**pac**E**) provides the ability for a user to prototype an experimental filesystem in user space, while allowing just that user to mount and access the filesystem. One way to solve this problem is to fork off a new namespace and allow the user to mount the experimental filesystem in it. This solves the problem; however, it also excludes the namespace from seeing any new mounts in the parent namespace. Ideally the user would want her own mounts to remain private to her namespace but be able to see the mounts in the parent namespace.

On LSPP systems, users need to be able to log in at various levels, and be able to use different contents in various directories depending on which level the user is logged in to the system. The solution to this is typically to use polyinstantiation. For example, each directory is actually several directories, and which one the user sees is determined by the privilege level of the user. For Linux this is implemented using filesystem-namespaces, the `unshare()` system call and Pluggable Authentication Module (PAM). But using filesystem-namespace restricts the user using a different namespace from seeing a newly inserted CD in the CD drive.

MVFS by IBM® provides multiple views of the same filesystem mount-tree. Depending on which view is used to access the files in the MVFS filesystem a different version of the file is visible. One way to implement this feature is to use filesystem-namespace. But this restricts a process from accessing two different filesystem-namespaces. Also the namespaces cannot be kept in sync across mount and unmount events. The other option is to mirror the filesystem mount-tree to different locations within the same filesystem mount-tree using the rbind feature. But this solution suffers from the problem that different versions of the mount-tree cannot be synchronized atomically when a new filesystem is mounted on one of the mirrors.

Virtualization products support multiple containers each with their own set of resources, and provide a individual view of the system. Processes are associated with a given container, and hence have to be visible only to processes within that container through the `procfs` interface. This demands maintaining multiple versions of `procfs`, each one corresponding to a container. One solution to this is to maintain multiple mirrors of the filesystem tree within the same filesystem, jailing processes in a given container under its corresponding mirror. The `procfs` filesystem associated with each mirror displays its virtualized world to that container. Again we face the same problem: how does a process in a container access the CD mounted outside its jail?

Shared subtree provides the mechanism that solves the above mentioned limitation. In Section 2, we describe the basic building blocks of shared-subtree. In Section 3, we explain the shared-subtree operational semantics. In Section 4, we discuss the implementation details in Linux. In Section 5 we review the applications.

2 Shared subtree semantics

As mentioned above, namespace provides isolation—processes in a namespace are protected from namespace-modifying operations done by processes outside. In other words, the

namespace boundary is a trust boundary. That is very useful in many situations—for instance, for bindings there is no analog of symlink vulnerabilities, simply because nobody outside our namespace is able to modify the bindings we see.

However, the same property sometimes becomes a hindrance because there is no way to arrange for modifications of parts of namespace short of sharing the namespace completely.

This situation is not entirely new; indeed, we have a similar though more simple problem with other kinds of possibly shared resource: the memory space. Processes are protected from each other and that is certainly a desirable thing. They also can share their entire memory space. However, it is often useful to have a *part* of memory space shared. That would appear to be a convenient model for our problem; however, it is not an exact fit.

It turns out to be more useful to speak not of sharing parts of the namespaces, but of propagating modifications among such parts. One of the chief reasons for that approach is that trust is not necessarily symmetric—allowing modifications done by process *A* to affect parts of the namespace of process *B* should not imply allowing the opposite.

In other words, the relationship "modifications to tree *X* cause corresponding modifications to tree *Y*" can not be reduced to "*X* and *Y* refer to the same shared entity" and we are better off treating that relationship as the first-class object.

The challenge, of course, is to provide coherent semantics for such propagation and implement it efficiently.

2.1 Definitions

Throughout the rest of the paper we will refer to operations modifying the mount trees as *mount events* or *propagation events*, whether they are made by `mount(2)` or by `umount(2)`.

To describe mount event propagation we need to introduce several new notions:

1. mounts are divided into *shared* and *solitary*.
2. shared mounts are partitioned into *peer groups*.[1] That controls the symmetric part of propagation.
3. The *propagation graph* controls the asymmetric part of propagation. It is a tree with peer groups and solitary mounts as nodes. All internal nodes are peer groups; only leaves may be solitary mounts.
4. In addition to that, some of the solitary mounts may be marked as *unbindable*.

We will refer to connected components of the propagation graph as *propagation trees*.

With respect to event propagation, there are four types of mounts:

1. shared mount
2. slave mount
3. private mount
4. unbindable mount

[1] Single-element peer groups are possible and do, in fact, play an important role. Their elements should not be confused with solitary mounts.

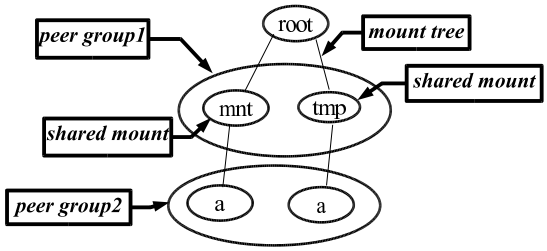

Figure 1: *shared-mount*

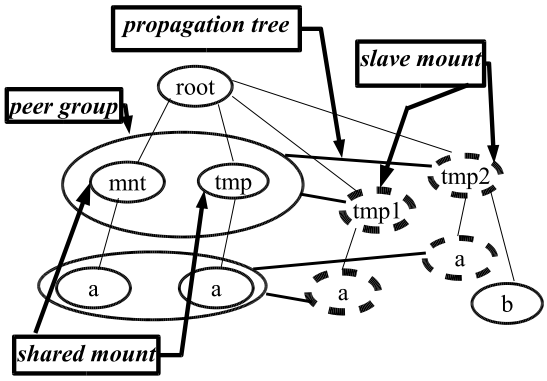

Figure 2: *slave-mount*

2.2 Shared-mount

As noted earlier, the current mount infrastructure lacks the ability to propagate mount events. Shared mounts provide the ability to keep several subtrees in sync; all events on a shared mount propagate to all its peers and new mounts created by such events will, in turn, become peers among themselves.

The new mount created from the shared-mount becomes a shared-mount too. And they together form members of the same peer group. A shared mount by itself is a sole member of its peer group. New clones of the shared-mount, inherit membership to the same peer group.

Figure 1 illustrates an example of shared mounts belonging to the same peer group. The mounts at `mnt` and `tmp` are shared, and belong to the same peer group *peer group1*. When a new filesystem *a* is mounted under `mnt`, the same filesystem automatically is mounted under `tmp` and these two mounts become members of a new peer group *peer group 2*.

The idea behind shared-mounts is being able to mount or unmount on any one of these mounts, and to have the action atomically propagated to all peers. The nice property of shared-mounts is that they allow mount-trees to remain identical across future mount and unmount.

2.3 Slave-mount

A slave-mount receives mount events from its master, but does not forward it back to its master. The master in this case is a peer group. Mounts of this type are preferred in cases where one would like to receive mounts in other namespaces, but would not like to share any mounts within its own namespace.

Figure 2 illustrates an example of a slave-mount. Note that *tmp1* and *tmp2* are slave mounts. When a new filesystem *a* is mounted under `mnt`, the same filesystem automatically is mounted under `tmp`, and also propagates to the slave mounts *tmp1* and *tmp2*. Whereas a mount of filesystem *b* on mount *tmp2* does not propagate anywhere.

2.4 Shared-and-slave-mount

A mount can be shared and slave at the same time. It would receive mount events from its master, share them with its peers, and possibly forward them to its slaves.

Note that all intermediate nodes in propagation trees would be peer groups consisting of mounts that are both shared and slaves; it is not something unusual.

2.5 Private-mount

A private-mount, as the name implies, does not carry any propagation semantics. It neither receives nor forwards any propagation events. Had there been no shared-subtree semantics, we would have only seen private-mounts.

2.6 Unbindable-mount

An unbindable-mount carries the same semantics as that of a private-mount. In addition, it disallows any of its contents including submounts from being mounted anywhere else.

3 Operational semantics of shared-subtree

In this section we define the interactions of various mount-related operations on the different flavors of mounts.

3.1 Mount operation

When a filesystem is mounted at a mountpoint, the behavior depends on the type of mount the mountpoint resides in.

If that mount is solitary, the filesystem is mounted at the mountpoint and created mount becomes private.

If the mount is a shared-mount, the filesystem is mounted at the mountpoint within the shared-mount, as well as at the corresponding locations in the peer-mounts and slave-mounts down the propagation-tree. Event propagation among the created mounts duplicates that among their parents.

3.2 Bind operation

The bind operation mounts a subtree of a filesystem directory tree on a mountpoint. The new mount inherits the properties of its source mount and as well as the properties of the destination mount on which it is mounted. The source mount is the mount containing the directory tree of the filesystem.

Table 1 indicates the semantics of the bind operation from a source mount, mounted on a destination mount.

The bind operation is invalid if the source mount is an unbindable mount.

The mount type is inherited from the source. If the source is shared, the new mount becomes its peer. If the source is a slave, the new mount becomes a slave of the same master.

If the mountpoint lies within a shared mount—i.e., the destination is a shared mount—the new mount becomes shared. Additional mounts are created at the corresponding locations in the peer mounts and slave mounts down the propagation tree. As in previous section, event propagation among the created mounts duplicates that among their parents.

3.3 Rbind operation

The rbind operation mounts a directory tree to a mountpoint. Unlike the bind operation, in the case of rbind, the source directory tree spans across mountpoints. The rbind operation behaves similar to bind operation, but if the source consists of more than one mount, the same actions apply to all of them.

If the source mount-tree contains any unbindable mounts, the rbind operation prunes off copies of the mount trees below such mounts before mounting them at new mountpoints.

source(A) ⇒ destination(B) ⇓	shared	private	slave	unbindable
shared	shared	shared	shared and slave	invalid
non-shared	shared	private	slave	invalid

Table 1: *The type of the new mount created when a source-mount A is bind mounted to a mountpoint residing in the destination mount B.*

3.4 Move operation

The move operation allows a mount tree to be moved to new mountpoint. Unlike bind or rbind operations, the source of the move operation must be a mountpoint. The operation is similar to the rbind operation.

If the destination mount containing the mountpoint is shared, the source mount becomes shared, too. Again, if the source mount was a slave, it becomes both shared and a slave. However, if the destination mount is solitary, the source-mount destination-mount is a non-shared mount; the source mount remains unchanged.

Note that a mount residing in a shared mount is not allowed to be moved.[2] Also, a mount tree containing an unbindable mount is disallowed from moving to a shared mount.[3]

Table 2 indicates the move semantics on a source mount to a mountpoint residing in a destination mount.

[2]If allowed, the move operation generates an unmount event. This unmounts all the mounts residing in other peer and slave mounts.

[3]If allowed, this operation will involve cloning unbindable mounts, which is disallowed. The author realizes that the mount tree could have been pruned below the unbindable mounts, while creating copies of the moved tree. This would provide semantics consistent with semantics of rbind.

3.5 Clone namespace operation

The *clone namespace* operation clones the entire mount tree of a namespace. All the mounts in the source namespace are cloned, and the resulting mount tree is associated with the new namespace. A copy of a shared mount becomes its peer, a copy of a slave becomes a slave of the same master. Note that this operation does clone the unbindable mounts.

3.6 Unmount operation

The unmount operation has subtle issues. Unmounting something mounted on a solitary mount is just a matter of removing the mount, provided that it is not being actively used and nothing is mounted on it.

Unmounting a mount X residing on a shared-mount P generates a propagation event. The mounts corresponding to X on all the mounts down the propagation tree of P are unmounted, unless there is something mounted on them. If some of these mounts are actively in use, the unmount fails.

Consider the mount tree shown in Figure 3: the mounts A, B, and C are all peers of each other. At the same time A, B, and C share a grandparent-parent-child relation. If unmount of C is attempted, since B is the parent of C, B generates an unmount propagation event

source(A) ⇒ destination(B) ⇓	shared	private	slave	unbindable
shared	shared	shared	shared and slave	invalid
non-shared	shared	private	slave	unbindable

Table 2: *The type of source-mount A when it is moved to a mountpoint residing in the destination mount B.*

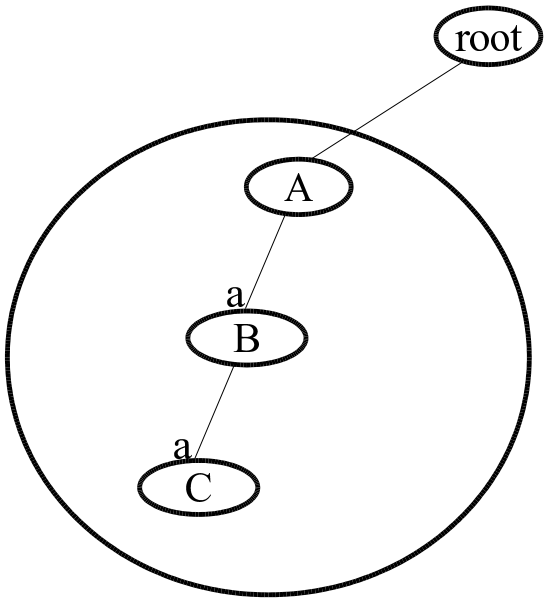

Figure 3: *A mount tree where the mounts in the same peer group also share a grandparent-parent-child relationship.*

that propagates to *A* and *C*. Since *B* is the child of *A* mounted at the same mountpoint as *C*, it has to be unmounted, too. But *B* cannot be unmounted because it has a sub-mount *C*. Hence the entire operation fails. This effectively makes the entire subtree under *A* unmountable.

To mitigate this problem we relaxed the unmount rule, by allowing unmount to succeed even if some of the mounts other than the one in question have sub-mounts.

3.7 Mount type transitions

A mount can transition through different types during its lifetime. During creation it acquires a state, depending on where it is created and from where it is cloned.

A user can explicitly transition the mount from any one type to any other according the shared-subtree semantics. Also the mount can change types implicitly when the mount is moved from one location to another as indicated in Table 2. Table 3 describes the type transition rules for a mount.

Note that an attempt to turn a shared mount that has no peers into a slave will make it private since there is no master to which it could be slaved to.

A solitary mount cannot be slaved.

3.8 Use of Unbindable-mount

The unbindable mounts are particularly useful to set up multiple identical mount trees within the same mount tree.

Figure 4 illustrates how the mount tree expands at each step as we create new copies of the mount tree. The unbindable mount contains the expansion by pruning off the subtree, thus creating exact copies of the mount tree at those locations.

	make-shared	make-slave	make-private	make-unbindable
shared	shared	shared/private	private	unbindable
slave	shared and slave	slave	private	unbindable
shared and slave	shared and slave	slave	private	unbindable
private	shared	private	private	unbindable
unbindable	shared	unbindable	private	unbindable

Table 3: *mount state transition*

3.9 Side-mounts

Shared subtree semantics can lead to peculiar situations. Suppose a mount *A* is the master of mount *B*. Mount *B* has a mount *C* on directory *b*. Suppose we mount *D* on directory *b* of mount *A*. The propagation event propagates to mount *B*. Should the new mount—let's say *E*, on mount *B* at directory *b*—be visible, or should it be obscured by the mount *C*? What happens when mount *D* is unmounted? Should mount *C* be unmounted or mount *E* be unmounted?

We define *side-mounts* as the sub-mounts on a given mount that are mounted on the same directory.

New mounts on the same directory of a mount are placed in a stack order, with the oldest mount always visible. An unmount request for a particular mount always unmounts the requested mount. However, unmounts triggered due to propagation always pop the most recent mount on the directory.

So in the example above, if an unmount of *C* is attempted, mount *C* is unmounted. However, if an unmount of *D* is attempted, *D* will be unmounted anyway, but the propagation event will unmount *E*, too (and not *C*).

4 Implementation Details

This section describes the changes made to data structures and the logic used to implement the shared-subtree feature in the Linux™ kernel.

4.1 Data Structure

The following four new fields were added to the `struct vfsmount` data structure to support the shared-subtree semantics.

1. `mnt_share`
2. `mnt_slave_list`
3. `mnt_slave`
4. `mnt_master`

`mnt_share` is a circular list of all the shared mounts that are peers of the given mount. `mnt_slave_list` is a circular list of all the slave mounts of the given mount. Mounts in all slave peer groups and slave mounts of a given mount are linked together in the mount's `mnt_share` circular list. `mnt_slave` runs through the circular list of all the slaves of the mount's master. `mnt_master` points to the master of the mount.

Figure 5 illustrates a data-structural representation of the shared-subtree.

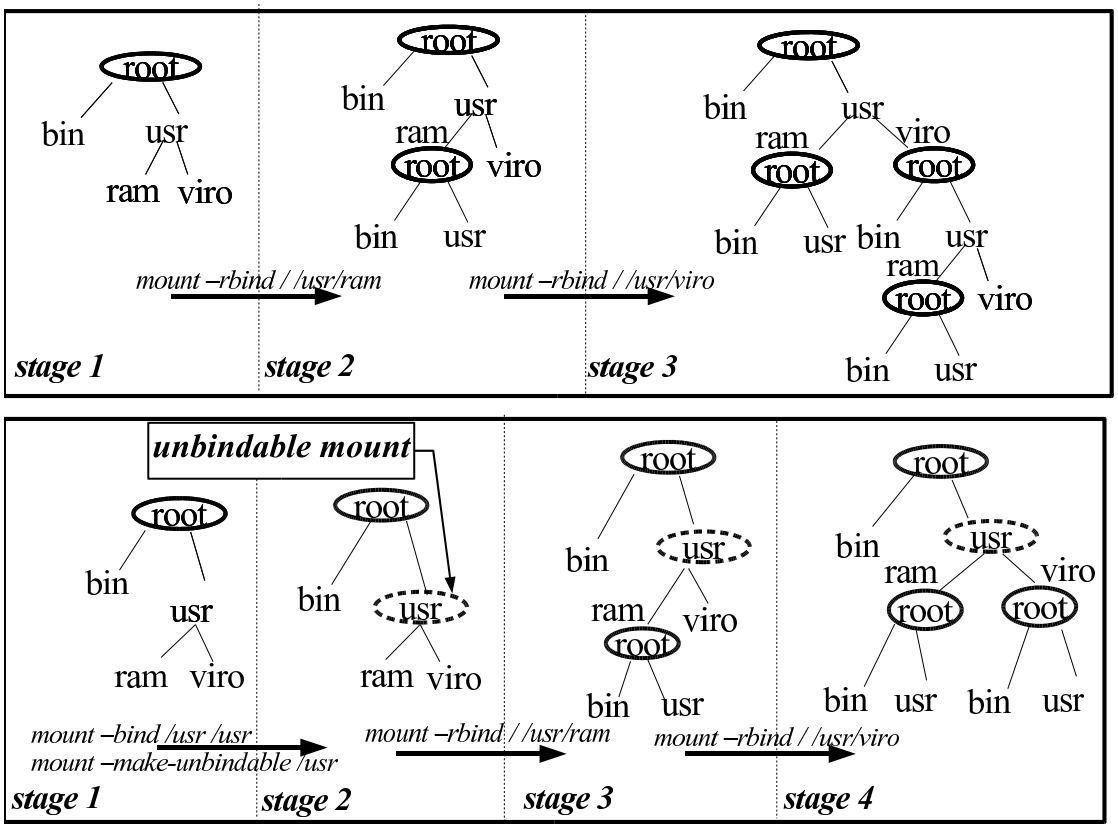

Figure 4:
Application of unbindable-mount

In the example we have the peer group P1, consisting of shared-mounts G1, G2, and G3. Peer-group P1 has three slave peer-groups: P2, P3, and P4. P2 consists of shared-mounts R1, R2, R3, and R4. Peer group P3 has M1 and M2. And peer-group P4 has Y1, Y2, Y3, and Y4 as its members. Peer group P1 has B2 as it slave-mount. And finally, peer-group P2 has O1 as its slave.

In our implementation we do not have an explicit data structure to represent a peer group. The peer group is implicitly managed by the circular list `mnt_share`. By walking the *mnt_share* circular list of a given mount, we find all the members of the peer group.

To find all the slaves of a peer group, we walk the circular lists represented by `mnt_slave_list` of all the mounts in the peer group. Note this circular list run through the `mnt_slave` field of the slave mounts.

The `mnt_master` field of a given mount points to the master mount of the slave-mount.

We introduced two new flags in the `mnt_flags` field of `struct vfsmount` to track the type of a given mount, namely `MNT_SHARE` and `MNT_UNBINDABLE`. These flags tell us if the mount is of type *shared* or *unbindable*. A non-NULL `mnt_master` indicates that the mount is of type *slave*. If the mount is not shared or slave or unbindable, then it is a private mount.

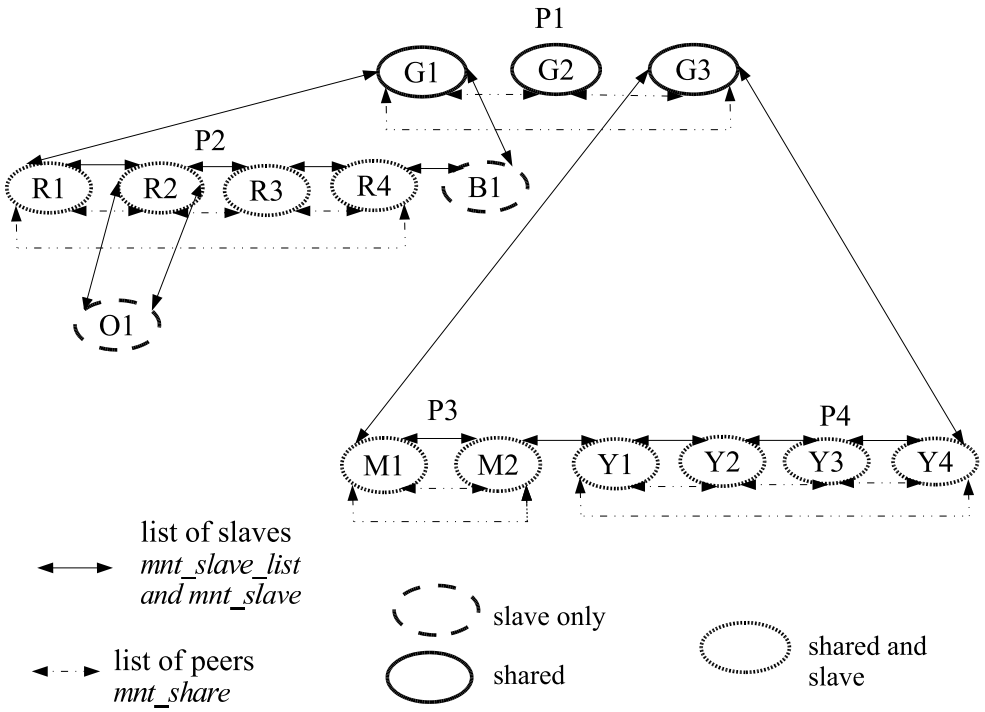

Figure 5: *Data-structure layout*

4.2 Description of mount, bind, move operations

The mount, unmount, bind, and move operations on shared mounts walk the propagation tree to collect all the affected mounts. Hence we have designed an efficient iterator `propagation_next()` that walks the propagation tree and returns the next mount in the propagation-tree. `propagation_next()` implements a depth-first walk returning the slave-mount first, followed by the peer mounts. As the tree is walked, new mounts can get linked into the propagation-tree.[4] This can happen during bind, move, or rbind operations. The iterator has enough information to skip these new mounts as it walks the propagation tree.

[4]If the new mountpoint is on a mount which already resides in the same propagation-tree.

A mount, bind, rbind, or move operation under a shared-mount typically involves creation of a number of copies of the mount tree that are to be mounted at the mountpoints where the mount operation propagates. `propagate_mnt()` walks the propagation tree of the destination mount, creating the necessary number of new mount trees. It returns a list of these mount trees. Also it ensures that all the mounts in each of the newly created mount trees are associated with their corresponding propagation trees. If creation of some mount tree fails, all the newly created copies are destroyed, in turn deleting them from their corresponding propagation trees. Note that the newly created mounts are attached to the propagation trees, but are not attached to the corresponding mount points. The caller of `propagate_mnt()` is responsible for attaching the created mount trees to their corresponding mountpoint atomically. We hold the `namespace_sem` semaphore during

the entire operation. That allows us to put the newly created mounts into the propagation trees immediately. `propagation_next()` skips these new mounts, and nobody else can walk the propagation trees until the semaphore is released.

The mountpoint may not exist in some of the mounts in a propagation tree.[5] This is typically the case when a subdirectory within a mount is bind-mounted. In cases where the mountpoint does not exist, our implementation still makes a copy of the mount tree, to be deleted later. One could implement an efficient way to avoid creation of this copy. Our implementation relies on the copy being around, because we use it to clone newer copies of the mount tree, these cloned copies being mounted on the slaves of the mount in question. The function `get_source()` tracks the copy of the mount tree to be used while cloning a new copy of the mount tree. `propagate_mnt()` keeps track of these extra copies of the mount tree. Once all the necessary mount trees are created, it deletes these defunct copies.

`attach_recursive_mnt()` takes the source mount tree and attaches it to the specified mountpoint in the destination mount. It uses `propagate_mnt()` to ensure that all the necessary mount trees are created successfully. On success, it atomically attaches all these mount trees at their corresponding destinations. In the case of a move operation, it also detaches the source mount tree from its parent before attaching to the new mountpoint.

[5]Consider a shared mount *A* having subdirectory *a* and *b*. When the subdirectory *a* is bind-mounted, a new shared mount *B* is created. If a new mount is attempted on directory *b* of mount *A*, the mount event though propagates to *B* will not have a mountpoint to mount.

4.3 namespace clone operation

All the mounts in the original namespace, including unbindable mounts, are cloned. The new mounts are attached to their corresponding propagation tree. If some allocations fail, the newly created mounts are detached from their propagation-tree and deleted.

4.4 Description of umount operation

The core of an unmount operation is in `umount_tree()` and `propagate_umount()`. All the mounts in the mount tree are first unhashed[6] and collected in a list. For each element in the list we run through its parent's propagation tree, to collect the corresponding child mounts. As explained in Section 3.6, we ignore the child mounts that contain submounts. Note that these mounts cannot be detached from the propagation tree while we are walking on it. Hence we detach them after we have completely collected all the mounts. At the same time, they are detached from their filesystem-namespaces. All these operations are done under the `vfsmount_lock` spinlock as well as the `namespace_sem` semaphore. The spinlock guards against races with other mount lookup routines. The semaphore guards against races with other mount and unmount operations. After all the mounts are unhashed, we release the spinlock. And after the semaphore is released, all the mounts are detached from their parent and are destroyed through `release_mounts()`.

[6]Unhashing a mount makes the mount inaccessible to any new lookups.

5 Applications of shared subtree

As noted in Section 1, the isolation property of filesystem namespace and the static nature of bind mounts restricts their usage in various applications.

The shared-subtree semantics solve these issues, and hence opens up the use of filesystem namespace and bind mounts to various applications which include SELinux's LSPP, MVFS, Virtualization, and FUSE, among others.

6 Future Work

The shared subtree semantics provide powerful constructs. These constructs help to solve problems faced by SELinux LSPP systems, the MVFS filesystem, and other projects.

Though this feature is efficiently supported by the kernel, currently the mount command[7] is unaware of shared-subtree semantics. As of this writing, a patch has been submitted.

It is easy to set up propagation trees and modify them. But no interface exists to display the setup of the propagation trees. It's impossible for a normal user to identify the type of a given mount without poking into the kernel data structures. A `procfs`- or `sysfs`-based interface that displays the propagation trees in some sane format is needed.

In many setups `/etc/mtab` is not guaranteed to match reality, which leads to enough confusion. Introduction of shared-subtree semantics aggravates the problem. It is very easy to contaminate `/etc/mtab` with non-existent mount entries. The mount command can never be aware of the new mounts created by the kernel due to propagation semantics. Although the `/proc/mounts` interface captures these new mounts, it ends up creating many identical entries. Also it does not capture all of the mount options. This entire mess warrants a clean and sane interface that would capture all the mount details.

Another problem is the lack of `vfsmount` accounting; if we are going to allow non-root mount, we will have to introduce some limits. Otherwise it's too easy to cause a DoS. It is not as simple as "who had called `mount(2)`," since we have to deal with extra slaves created by user `rbind` with the master being created by `root` and later mounted upon by `root`.

It is easy to create setups allowing, e.g., passing mounts between login sessions of a given user while allowing him to have private namespaces. That in itself does not require any kernel modifications—it's a matter of policy and can be easily arranged by userland. Moreover, it's easy to arrange for user-controlled export of parts of its namespace to other (willing) users, with no action required by sysadmin. E.g., it can be achieved by having `/share` shared in the first namespace and `sshd` doing the following:

1. create a new namespace
2. bind `/share/$USER` to `~/share`
3. for each pair `($who, $what)` such that `/share/$USER/$who/$what` exists, look in `/share/$who/allowed` for `peer $what $USER` or `slave $what $USER`. If the former is found, rbind `/share/$who/$what` on `/share/$USER/$who/$what`; if the latter is found, do the same and follow with marking the subtree under `/share/$USER/$who/$what` as slave.
4. rbind `/share/$USER` to `~/share`

[7] Packaged in `util-linux`.

5. mark subtree under `/share` as private.
6. `umount -l /share`

That provides `~/share` as shared between all sessions and allows exporting its parts to other users. All control over such exports and imports is done by users themselves: no sysadmin intervention is required. `libpam` is the obvious place for such functionality.

Unfortunately, this and similar schemes exacerbate the need of non-root bindings. With those we are firmly in the territory where modifying namespace becomes a useful operation outside of system setup context. In other words, that's where we run into the need of properly done mount accounting. Implementation should be relatively straightforward, but we need to get the semantics right and verify that it takes care of all corner cases.

7 Acknowledgement

We would like to thank the following people for their help with this paper. Serge Hallyn and Dave Hansen for providing input to this paper. Nishanth Aravamudhan and Balbir Singh for helping with the figures. We would also like to thank Mike Waychison, Miklos Szeredi, and Bruce J. Fields for their feedback and input during implementation of Shared-Subtree. To Avantika Mathur for providing the shared-subtree test suite. It is very likely that there are others who we have inadvertently failed to acknowledge; for you, we apologize for the omission and thank you for you efforts.

8 Legal Statement