Improving the Linux mount utility

Adding support for new functionality of the mount syscall

Dirk Gerrits René Gabriëls Peter Kooijmans

April 20, 2007

2DI40 The Linux Operating System 2006/2007
Department of Mathematics & Computer Science
Technische Universiteit Eindhoven

0531594	René Gabriëls	<r.gabriels@student.tue.nl>
0531798	Dirk Gerrits	<dirk@dirkgerrits.com>
0538533	Peter Kooijmans	<p.j.a.m.kooijmans@student.tue.nl>

Contents

1 Introduction **2**

2 Mounting in Linux **2**

2.1 Mount utility 2

2.2 Mount system call 3

3 Discrepancies **4**

4 Relative access time **5**

5 POSIX access control lists **6**

6 Shared subtrees **7**

6.1 An overview 7

6.2 Recursion 9

6.3 Our implementation 9

6.4 A bug in the stock Linux kernel 12

A Source code patches **14**

1 Introduction

The assignment was to add missing command line options to the mount utility (part of the `util-linux` package), such that it was in sync with the mount system call of the Linux kernel. If any bugs in existing code were discovered, they would have to be fixed. This report describes the discrepancies we discovered between the mount utility and the system call, and the changes we made to the mount utility and to the Linux kernel to correct them. The actual code patches are listed in the appendix.

2 Mounting in Linux

Traditionally, the Linux mount utility is used to mount a device containing a file system on a directory in the virtual file system. The umount utility is used to do the reverse operation: remove a mounted filesystem from the virtual file system. These utilities are mostly wrappers around the system calls mount and umount respectively.

The list of current mounts is managed by both the kernel (accessible via the file `/proc/mounts`) and the mount utility (accessible via the file `/etc/mtab`).

2.1 Mount utility

The Linux mount utility requires three command line arguments: a device to mount, a directory under which the device should be mounted, and the filesystem type that is present on the device. Additionally, some general and file system specific options may be specified. An example:

```
# mount -t reiserfs -o nolog /dev/sda2 /mnt/test/
```

(Here `/dev/sda2` signifies the device, `/mnt/test/` the directory, and `reiserfs` the filesystem type.)

All known devices and their filesystem and mount point can be specified in the file `/etc/fstab`, such that they can be mounted by specifying to mount only the device or the mount point. Furthermore, this file is used to specify what should be mounted at boot time.

The umount utility can reverse the effect of the mount utility: remove a mounted device from the filesystem tree. For example:

```
# umount /mnt/test/
```

The mount and umount utilities maintain a list of mounts at `/etc/mtab`. This enables the user to list the current mounts, and enables umount to free any resources reserved by mount (e.g. loopback devices).

Since Linux 2.4.0, it is possible to move and copy existing mounts. Moving a mount point `a` to another mount point `b`, simply moves the entire VFS-tree rooted at `a` to `b`, thereby shadowing the old mount at `b`. Copying (known as binding) from a mount point `a` to a mount point `b`, mounts at `b` whatever is mounted at `a`, thereby shadowing the old mount at `b`. Copying can also be done recursively (known as recursive bind), meaning that all submounts of `a` appear under `b` as well.

These operations have been implemented by the mount utility, but do not fit in the existing argument scheme at the time. A new syntax was devised: first specify the action (either `--move`, `--bind` or `--rbind`) and then specify the source and destination. For example:

```
# mount -t reiserfs /dev/sda2 /mnt/test1/
# mount --bind /mnt/test1 /mnt/test2
# umount /mnt/test1
# umount /mnt/test2
```

2.2 Mount system call

The mount system call `sys_mount` is defined in `fs/namespace.c`:

```
asmlinkage long sys_mount(char __user * dev_name, char __user * dir_name,
                          char __user * type, unsigned long flags,
                          void __user * data)
```

The parameters are:

1. `dev_name`: the device on which the filesystem to be mounted is present.
2. `dir_name`: the directory under which the filesystem must be mounted.
3. `type`: the filesystem type
4. `flags`: the global mount flags (a bitfield).
5. `data`: file system specific options.

This corresponds to the classical mount command, except that the file system specific options are specified by the parameter `data`. Binding and moving are implemented by specifying the source in the `dev_name` parameter, the destination in the `dir_name` parameter, leaving the filesystem type empty and setting the proper flags in the parameter `flags`.

3 Discrepancies

The following table lists all options that the kernel supports, those that the old mount utility supports, those that the new mount utility supports, and the command line parameters required to set those options. The new options labeled with a * have been added to the mount utility by us. This list was extracted from the source code of the kernel and the mount utility.

Bit	Kernel	Mount (old)	Mount (new)	Option
0	MS_RDONLY	MS_RDONLY	MS_RDONLY	-o rw / ro
1	MS_NOSUID	MS_NOSUID	MS_NOSUID	-o suid / nosuid
2	MS_NODEV	MS_NODEV	MS_NODEV	-o dev / nodev
3	MS_NOEXEC	MS_NOEXEC	MS_NOEXEC	-o exec / noexec
4	MS_SYNCHRONOUS	MS_SYNCHRONOUS	MS_SYNCHRONOUS	-o sync / async
5	MS_REMOUNT	MS_REMOUNT	MS_REMOUNT	-o remount
6	MS_MANDLOCK	MS_MANDLOCK	MS_MANDLOCK	-o mand / nomand
7	MS_DIRSYNC	MS_DIRSYNC	MS_DIRSYNC	-o dirsync
8		MS_AFTER	MS_AFTER	--after
9		MS_OVER	MS_OVER	--over
10	MS_NOATIME	MS_NOATIME	MS_NOATIME	-o atime / noatime
11	MS_NODIRATIME	MS_NODIRATIME	MS_NODIRATIME	-o diratime / nodiratime
12	MS_BIND	MS_BIND	MS_BIND	--(r)bind
13	MS_MOVE	MS_MOVE	MS_MOVE	--move
14	MS_REC	MS_REC	MS_REC	(the “r” in other options)
15	MS_SILENT	MS_VERBOSE	MS_VERBOSE	
16	MS_POSIXACL	MS_LOOP		
17	MS_UNBINDABLE	MS_COMMENT	MS_UNBINDABLE*	--make-(r)unbindable
18	MS_PRIVATE		MS_PRIVATE*	--make-(r)private
19	MS_SLAVE		MS_SLAVE*	--make-(r)slave
20	MS_SHARED		MS_SHARED*	--make-(r)shared
21	MS_RELATIME		MS_RELATIME *	-o relatime / norelatime
22			MS_LOOP	-o loop=*device*
23			MS_COMMENT	-o _netdev
24				
25				
26				
27		MS_GROUP	MS_GROUP	-o group / nogroup
28		MS_OWNER	MS_OWNER	-o owner / noowner
29		MS_USER	MS_USER	-o user / nouser
30	MS_ACTIVE	MS_USERS	MS_USERS	-o users / nousers
31	MS_NOUSER	MS_NOAUTO	MS_NOAUTO	-o auto / noauto

MS_AFTER and MS_OVER are defined in the mount utility, but not in the kernel. Although this isn't documented in the man pages, these options can still be passed to the mount utility. Doing so, however, doesn't really accomplish anything.

The deprecated MS_VERBOSE, and the new MS_SILENT taking its place have some supporting

code in the mount utility (-o quiet / loud), but this code is disabled by compile-time conditionals. The -v flag that is supported instead enables verbosity in the mount utility itself (rather than the system call), and will generally be more useful to users.

The MS_LOOP and MS_COMMENT options are implemented in the mount utility entirely, without explicit kernel support.

The MS_NOUSER and MS_ACTIVE options are for kernel-use only and shouldn't be implemented in the mount utility. MS_NOUSER specifies that the filesystem must never be mounted from user space. (Used in rootfs, a special instance of ramfs.) MS_ACTIVE is also only used internally, and serves to prevent certain race conditions involving iput.

The option -o relatime is implemented by the kernel, but wasn't supported by the mount utility. Similarly, the actions --make-(r)unbindable, --make-(r)private, make-(r)slave, --make-(r)shared are defined in the kernel, but weren't implemented by the mount utility. In the next sections our implementation of all these new parameters will be discussed. We also explain why we didn't add a -o posixacl option for the kernel's MS_POSIXACL.

4 Relative access time

The kernel provides the "relative access time" option (MS_RELATIME), which is similar to the "no access time" option (MS_NOATIME), but not yet available through the mount utility. Relative access time means that the access time of an inode is only updated if it's less than or equal to the modify or change time of that inode. This is useful to improve performance.

An example:

```
# mkdir /mnt/test
# mount -o relatime /dev/sda2 /mnt/test
# cat /etc/mtab | grep relatime
/dev/sda2 /mnt/test reiserfs rw,relatime 0 0
# touch /mnt/test/foo
# stat /mnt/test/foo
  File: '/mnt/test/foo'
  Size: 0               Blocks: 0          IO Block: 4096   regular empty file
Device: 802h/2050d      Inode: 8874424     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-04-16 13:56:49.000000000 +0200
Modify: 2007-04-16 13:56:49.000000000 +0200
Change: 2007-04-16 13:56:49.000000000 +0200
# echo "bar" > /mnt/test/foo
# stat /mnt/test/foo
  File: '/mnt/test/foo'
  Size: 4               Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d      Inode: 8874424     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-04-16 13:56:49.000000000 +0200
```

```
Modify: 2007-04-16 13:57:43.000000000 +0200
Change: 2007-04-16 13:57:43.000000000 +0200
# cat /mnt/test/foo
bar
# stat /mnt/test/foo
  File: '/mnt/test/foo'
  Size: 4               Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d      Inode: 8874424     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-04-16 13:58:03.000000000 +0200
Modify: 2007-04-16 13:57:43.000000000 +0200
Change: 2007-04-16 13:57:43.000000000 +0200
mount # cat /mnt/test/foo
bar
mount # stat /mnt/test/foo
  File: '/mnt/test/foo'
  Size: 4               Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d      Inode: 8874424     Links: 1
Access: (0644/-rw-r--r--)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2007-04-16 13:58:03.000000000 +0200
Modify: 2007-04-16 13:57:43.000000000 +0200
Change: 2007-04-16 13:57:43.000000000 +0200
```

So the access time is only updated when the first `cat` is executed, and will only be updated again after a modification is made, or the filesystem is remounted with this option disabled.

5 POSIX access control lists

Since the 2.6 kernel, some filesystems support POSIX access control lists as a replacement for the traditional UNIX access control bits. In the traditional UNIX model, read, write and execute permissions can be set for the file owner, for a specific group and for all others. With access control lists it is possible to extend this to more than one user or group. The kernel supports this feature through the `MS_POSIXACL` mount system call option.

Currently, not all filesystems support access control lists. Grepping through the kernel source code for the `MS_POSIXACL` flags reveals the following. JFS, JFFS2 and CIFS enable access control lists by default if this turned on in the kernel's configuration. ReiserFS and GFS2 have support for access control lists, but not through the standard kernel flag. These filesystems need the "acl" option to be in the additional options for the mount system call to enable this feature. Ext2, Ext3 and Ext4 allow access control lists to be enabled either through the standard kernel flag, or the extra option "acl". The NFS filesystem turns the `MS_POSIXACL` option on by default, keeping file-system permission handling to itself, instead of letting the Linux kernel handle it.

These examples show that support of the `MS_POSIXACL` flag is restricted to only a few filesystems and even they already have an alternate way of allowing access control lists to be enabled

through the mount utility. That's why we have decided not to implement this flag, at least not until the option is more universally supported by filesystems.

6 Shared subtrees

Since the 2.6.15 kernel the mount system call has had support for *shared subtrees*. Before, performing a bind operation always made an independent copy of the subtree. With this new functionality, bound subtrees can propagate new submounts and -unmounts to each other even *after* the bind.

At the time of this writing (util-linux version 2.12r), support for this functionality hasn't made its way into the mount user-space utility yet. In the explanation below we're going to assume that it has, discussing how we added it afterwards.

6.1 An overview

There are 4 states a mount point can be in, and its state affects the way a subsequent bind with this mount point as the source will work:

- Private. (This is the default state.)

  Private mount points act as per the pre-2.6.15 semantics: a bind is a simple copy and no new events will propagate between the bind's source and the destination afterwards.

  If a mount point `a` is in another state (see below how to accomplish that), it can be made private again with:

  ```
  # mount --make-private a
  ```

- Shared.

  When a shared mount point is bound to another mount point, subsequent mount and unmounts events under either will propagate to the other.

  For example, suppose that `a` names a mount point. Then we could make it shared and bind it to `b` as follows:

  ```
  # mount --make-shared a
  # mount --bind a b
  ```

  Both `a` and `b` are now in the shared state, and anything we (un)mount under either `a` or `b` will be (un)mounted under both:

  ```
  # mkdir a/p1 b/p2
  # mount /dev/sd0 a/p1
  # ls a/p1
  d1 d2 d3
  ```

```
# ls b/p1
d1 d2 d3
# mount /dev/sd1 b/p2
# ls b/p2
e1 e2
# ls a/p2
e1 e2
# umount b/p1
# ls a/p1
# umount a/p2
# ls b/p2
```

In this example, `a` and `b` form a *peer group*: a group of mount points that propagate events to each other.

- Slave.

  A slave mount point is similar to a shared mount point, but events are only propagated towards it, not from it. A mount point can only be made a slave if it is part of a peer group first. It then becomes slave to all its former peers, receiving propagation events *from* them, but not sending its own *to* them.

  Continuing the example above, we could make `a` a slave to `b` as follows:

```
# mount --make-slave a
```

  Then, any (un)mounting under `b` would show up under `a` but not vice versa:

```
# mount /dev/sd0 a/p1
# ls a/p1
d1 d2 d3
# ls b/p1
# mount /dev/sd1 b/p2
# ls b/p2
e1 e2
# ls a/p2
e1 e2
# umount b/p1
umount: b/p1: not mounted
# umount a/p1
# umount a/p2
# ls b/p2
e1  e2
# umount b/p2
```

  Note that we could have just as well made `b` a slave to `a` instead, in the exact same fashion. That `a` was the source and `b` the destination of the bind that set up their peer group is of no relevance.

- Unbindable.

  An unbindable mount point is simply one that cannot be the source of a bind operation:

  ```
  # mount --make-unbindable a
  # mount --bind a b
  mount: wrong fs type, bad option, bad superblock on a,
         missing codepage or other error
         In some cases useful info is found in syslog - try
         dmesg | tail  or so
  ```

  Note that both `--make-private` and `--make-unbindable` sever the propagation links of the target mount point to and from any peers it has.

Apart from these 4 states there's actually (more or less) a fifth state. When a mount point is made a slave to its peers, it can then be made shared again with `--make-shared`. Such a mount point is said to be in the "shared and slave" state.

The mount point will still be slave to the peers it had at the time of the `--make-slave`, thus receiving propagation events from them, but not sending them any. However, it can now also get new peers (through bind) with which it will have two-way propagation. (And by "transitivity of slavery" its new peers will receive events from its old peers, but not the other way around.)

6.2 Recursion

All of `--make-private`, `--make-shared`, `--make-slave`, and `--make-unbindable` have recursive versions (respectively: `--make-rprivate`, `--make-rshared`, `--make-rslave`, and `--make-runbindable`). These will not just try to change the state of the target mount point, but also of any submounts it has at that point.

The `--bind` operation also has a recursive version (`--rbind`), which it has had for quite some time. It continues to operate the way it has: as if a `--bind` was performed on each submount separately until the entire subtree is replicated. The only caveat here is if any of those submounts is unbindable. In that case `--rbind` will prune any subtrees rooted in an unbindable submount. Figure 1 shows an example of the result of `mount --rbind A Z` where `A` has an unbindable submount `C`.

6.3 Our implementation

New command-line options

The easy part of our modification to the mount utility was to add the `--make-(r)private`, `--make-(r)shared`, `--make-(r)slave`, and `--make-(r)unbindable` command line options, translating them into `MS_PRIVATE(|MS_REC)`, `MS_SHARED(|MS_REC)`, `MS_SLAVE(|MS_REC)`, and `MS_UNBINDABLE(|MS_REC)` flags for the kernel system call.

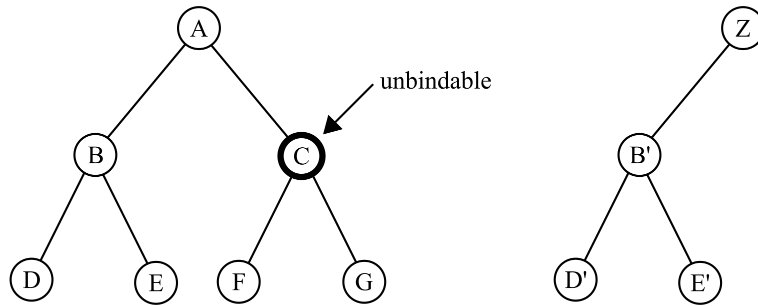

Figure 1: Example of the result of an `--rbind` with an unbindable submount in the source tree.

Fixing `/etc/mtab` for `--bind` and `--move`

More complex was to make `/etc/mtab` behave in a "reasonable" way, for both the user reading the file, and the (u)mount utility doing the same. First we set out to fix the way `mount --bind` and `mount --move` updated `/etc/mtab`.

In the old mount utility, `mount --bind a b` simply wrote a line in `/etc/mtab` specifying that the "device" `a` got mounted on directory `b`. While this gives the user "some" information as to what is mounted under `b`, it does make umount behave unintuitively. For example:

```
# mount --bind a a
# mount --bind a b
# umount a
# umount a
# umount b
umount: b: not mounted
```

Here we unmounted `b` by unmounting `a` twice! The reason that happens is that the first `umount a` finds `a` in `/etc/mtab` as a directory, while the second finds it again as the device mounted under directory `umount b`.

This behaviour of umount to allow the user to unmount by specifying a device is rather archaic. It has been possible to have the same device mounted on several locations for a very long time. However, for simple situations it can be convenient to the user, so we didn't want to change this behaviour.

Instead we made `mount --bind a b` never write a line in `/etc/mtab` specifying `a` as a device. Instead we make it look for the device mounted under `a` in `/etc/mtab` and then we write a line stating that this device is also mounted under `b`. In effect, `a`'s entry in `/etc/mtab` is copied with the directory field replaced by `b`. In case `a` wasn't found, the device for `b` is set to "`none`" instead.

For `mount --move a b` we made a similar change, except that `a`'s line is replaced instead of copied.

Shared subtrees in `/etc/mtab`

In addition to the changes for `--bind` and `--move` in the `/etc/mtab` behaviour, we also made our new operations update it. We made `mount --make-shared/slave/unbindable a` put "`shared/slave/unbindable`" in the options field for `a`'s line in `/etc/mtab`. The `--make-private` action puts nothing (extra) in the options field, so "privateness" is indicated by the lack of `shared`, `slave`, and `unbindable` in the options field.

In this we made sure none of these keywords ever appears more than once, or together with any of the others so that a mount point doesn't appear to the user as being in multiple states. We also make the mount utility emit a warning when `a` wasn't found in `/etc/mtab`, stating that we are unable to update `/etc/mtab` accordingly:

```
# mount -n --bind a a
# mount --make-unbindable a
mount: warning: couldn't update /etc/mtab, because no previous entry was \
found for /root/a
```

We believe this behaviour is reasonable, but it is certainly not correct in every situation. For example, `--make-slave` will always write "`slave`" in the options field of the mount point, but the mount point won't actually become a slave unless it was part of some peer group beforehand.

As another example, `mount --bind a b` with `a` being private and `b` being shared will add a line stating that `b` is private, while it really is shared.

Also, performing any of the recursive operations will only create or update one line in `/etc/mtab`, rather than one for each affected submount.

There is no foolproof way around these problems because the system call doesn't provide us with all the needed information. Even if we could assume we have access to `/proc/mounts` (which we can't), then we don't know enough: the state of mount points isn't shown there.

Creating shared subtrees through `/etc/fstab`

The described changes also carry over to `/etc/fstab`. That is, one can specify shared subtrees in `/etc/fstab` and use `mount -a` (at boot time or otherwise) to have them created.

A line in `/etc/fstab` consists of 6 whitespace-separated fields: device, directory, filesystem type, options, "dump", and "pass". For shared subtrees you'll only use the device and directory, and the options field is used to specify the operation. The other fields can be left "none" or "0". Here's an example of some "manual" mount command lines, and how they can be specified in `/etc/fstab` to be automated:

```
shell: mount --make-shared /mnt/a
fstab: none /mnt/a none make-shared 0 0
```

```
shell: mount --bind /mnt/a /mnt/b
fstab: /mnt/a /mnt/b none bind 0 0

shell: mount --make-slave /mnt/b
fstab: none /mnt/b none make-slave 0 0

shell: mount --move /mnt/c /mnt/b
fstab: /mnt/c /mnt/b none move 0 0
```

Note that these operations are mutually exclusive. One can't write, for example, "`/mnt/a /mnt/b none bind,make-slave 0 0`" to try to bind `/mnt/a` to `/mnt/b` and then make `/mnt/b` a slave. Two separate lines are needed for that.

This is because the mount utility parses all options into bitflags, ORs them together (destroying their ordering), and then passes that on to the system call which will only perform the action whose bitpattern is the first to match (in this case `bind`), or fails in a less intuitive way (for example "`make-shared,make-unbindable`" would cause a `make-private`).

Another caveat is that the mount utility only reads in `/etc/mtab` once at the start, so during a `mount -a` it won't see the updates it has already made to the file. This means that a `bind` or `move` won't always be able to copy the correct information, nor will `make-`{`private`, `shared`, `slave`, `unbindable`} always be able to correctly update the right entry. This limitation could be worked around, but we deemed it more trouble than it's worth.

6.4 A bug in the stock Linux kernel

As of this writing (2.6.20 kernel) the mount system call contains a bug where the `--make-private` operation does not reset the unbindable bit:

```
# mount --make-unbindable a
# mount --bind a b
mount: wrong fs type, bad option, bad superblock on a,
       missing codepage or other error
       In some cases useful info is found in syslog - try
       dmesg | tail  or so
# mount --make-private a
# mount --bind a b
mount: wrong fs type, bad option, bad superblock on a,
       missing codepage or other error
       In some cases useful info is found in syslog - try
       dmesg | tail  or so
```

A workaround is to perform a `--make-shared` first to reset the unbindable bit, and then a subsequent `--make-private` to reset the shared bit and arrive at the desired state.

We could have opted to add this workaround behaviour to the user space mount utility, but we instead decided to rely on the kernel to be fixed in the near future. We wrote and tested a patch ourselves, which merely adds two lines to `change_mnt_propagation` in `fs/pnode.c`:

```
void change_mnt_propagation(struct vfsmount *mnt, int type)
{
    if (type == MS_SHARED) {
        set_mnt_shared(mnt);
        return;
    }
    do_make_slave(mnt);
    if (type != MS_SLAVE) {
        list_del_init(&mnt->mnt_slave);
        mnt->mnt_master = NULL;
        if (type == MS_UNBINDABLE)
            mnt->mnt_flags |= MNT_UNBINDABLE;
        else                                   /* ADDED */
            mnt->mnt_flags &= ~MNT_UNBINDABLE; /* ADDED */
    }
}
```

A Source code patches

Listing 1: kernel patch

```
--- pnode.old    2007-04-17 12:53:11.000000000 +0200
+++ pnode.c      2007-04-17 13:22:03.000000000 +0200
@@ -83,6 +83,8 @@
                 mnt->mnt_master = NULL;
                 if (type == MS_UNBINDABLE)
                         mnt->mnt_flags |= MNT_UNBINDABLE;
+                else
+                        mnt->mnt_flags &= ~MNT_UNBINDABLE;
         }
 }
```

Listing 2: mount_constants.h patch

```
--- mount_constants.h.old       2007-04-17 12:01:34.000000000 +0200
+++ mount_constants.h    2007-04-20 10:45:30.000000000 +0200
@@ -66,3 +66,19 @@
 #ifndef MS_MGC_MSK
 #define MS_MGC_MSK 0xffff0000   /* magic flag number mask */
 #endif
+
+#ifndef MS_UNBINDABLE
+#define MS_UNBINDABLE  (1<<17) /* change to unbindable */
+#endif
+#ifndef MS_PRIVATE
+#define MS_PRIVATE     (1<<18) /* change to private */
+#endif
+#ifndef MS_SLAVE
+#define MS_SLAVE       (1<<19) /* change to slave */
+#endif
+#ifndef MS_SHARED
+#define MS_SHARED      (1<<20) /* change to shared */
+#endif
+#ifndef MS_RELATIME
+#define MS_RELATIME    (1<<21) /* Update atime only when smaller than mtime/ctime. */
+#endif
```

Listing 3: mount.c patch

```
--- mount.c.old 2007-04-17 12:01:15.000000000 +0200
+++ mount.c     2007-04-20 12:45:39.000000000 +0200
@@ -74,7 +74,9 @@
 /* Add volumelabel in a listing of mounted devices (-l). */
 static int list_with_volumelabel = 0;

-/* Nonzero for mount {--bind|--replace|--before|--after|--over|--move} */
+/* Nonzero for mount {--bind|--replace|--before|--after|--over|--move|
+                      --make-unbindable|--make-private|--make-slave|
+                      --make-shared} */
 static int mounttype = 0;

 /* True if ruid != euid.  */
@@ -98,8 +100,8 @@
 #define MS_USER        0x20000000
 #define MS_OWNER       0x10000000
 #define MS_GROUP       0x08000000
-#define MS_COMMENT     0x00020000
-#define MS_LOOP        0x00010000
+#define MS_COMMENT     0x00800000
+#define MS_LOOP        0x00400000
```

```
 /* Options that we keep the mount system call from seeing.  */
 #define MS_NOSYS         (MS_NOAUTO|MS_USERS|MS_USER|MS_COMMENT|MS_LOOP)
@@ -127,6 +129,7 @@
   { "async",    0, 1, MS_SYNCHRONOUS},  /* asynchronous I/O */
   { "dirsync", 0, 0, MS_DIRSYNC},      /* synchronous directory modifications */
   { "remount",  0, 0, MS_REMOUNT},      /* Alter flags of mounted FS */
+  { "move",     0, 0, MS_MOVE    },     /* Relocate part of tree elsewhere */
   { "bind",     0, 0, MS_BIND    },     /* Remount part of tree elsewhere */
   { "rbind",    0, 0, MS_BIND|MS_REC }, /* Idem, plus mounted subtrees */
   { "auto",     0, 1, MS_NOAUTO },      /* Can be mounted using -a */
@@ -164,6 +167,26 @@
   { "diratime",        0, 1, MS_NODIRATIME },  /* Update dir access times */
   { "nodiratime", 0, 0, MS_NODIRATIME },/* Do not update dir access times */
 #endif
+#ifdef MS_UNBINDABLE
+  { "make-unbindable", 0, 0, MS_UNBINDABLE }, /* Change to unbindable */
+  { "make-runbindable", 0, 0, MS_UNBINDABLE|MS_REC }, /* recursively */
+#endif
+#ifdef MS_PRIVATE
+  { "make-private", 0, 0, MS_PRIVATE }, /* Change to private */
+  { "make-rprivate", 0, 0, MS_PRIVATE|MS_REC }, /* recursively */
+#endif
+#ifdef MS_SLAVE
+  { "make-slave", 0, 0, MS_SLAVE }, /* Change to slave */
+  { "make-rslave", 0, 0, MS_SLAVE|MS_REC }, /* recursively */
+#endif
+#ifdef MS_SHARED
+  { "make-shared", 0, 0, MS_SHARED }, /* Change to shared */
+  { "make-rshared", 0, 0, MS_SHARED|MS_REC }, /* recursively */
+#endif
+#ifdef MS_RELATIME
+  { "relatime",   0, 0, MS_RELATIME }, /* Update atime when < mtime/ctime */
+  { "norelatime", 0, 1, MS_RELATIME }, /* Update atime always */
+#endif
   { NULL,        0, 0, 0          }
 };

@@ -474,7 +497,8 @@
     if (*types && strcasecmp (*types, "auto") == 0)
        *types = NULL;

-    if (!*types && (flags & (MS_BIND | MS_MOVE)))
+    if (!*types && (flags & (MS_BIND | MS_MOVE | MS_UNBINDABLE | MS_PRIVATE |
+                              MS_SLAVE | MS_SHARED)))
        *types = "none";          /* random, but not "bind" */

    if (!*types && !(flags & MS_REMOUNT)) {
@@ -642,10 +666,48 @@
   return 0;
 }

+static char *
+update_shared_subtree_options(const char *old_opts, int flags) {
+        char *new_flag = "", *new_opts, *d;
+        const char *s;
+
+        if (flags & MS_UNBINDABLE)
+                new_flag = ",unbindable";
+        else if (flags & MS_PRIVATE)
+                new_flag = "";
+        else if (flags & MS_SLAVE)
+                new_flag = ",slave";
+        else if (flags & MS_SHARED)
+                new_flag = ",shared";
+
+        new_opts = (char*)xmalloc(strlen(old_opts)+strlen(new_flag)+1);
```

```
+
+        /* Copy the old options, apart from any shared subtree stuff. */
+        d = new_opts; s = old_opts;
+        while (*s) {
+                if (strcmp(s, "unbindable") == 0 || strcmp(s, "slave") == 0 ||
+                    strcmp(s, "shared") == 0) {
+                        while (*s != ',' && *s != '\0') s++;
+                        if (*s == ',') s++;
+                } else {
+                        while (*s != ',' && *s != '\0') *d++ = *s++;
+                        if (*s == ',') { *d++ = *s++; }
+                }
+        }
+        if (d != new_opts && *(d-1) == ',') --d;
+        *d = '\0';
+
+        /* Add the new shared subtree option. */
+        strcat(new_opts, new_flag);
+
+        return new_opts;
+}
+
 static void
 update_mtab_entry(const char *spec, const char *node, const char *type,
                   const char *opts, int flags, int freq, int pass) {
         struct my_mntent mnt;
+        struct mntentchn *old;

         mnt.mnt_fsname = canonicalize (spec);
         mnt.mnt_dir = canonicalize (node);
@@ -662,9 +724,38 @@
         if (!nomtab && mtab_is_writable()) {
                 if (flags & MS_REMOUNT)
                         update_mtab (mnt.mnt_dir, &mnt);
-                else {
+                else if (flags & (MS_UNBINDABLE|MS_PRIVATE|MS_SLAVE|MS_SHARED)) {
+                        old = getmntdirbackward(mnt.mnt_dir, NULL);
+                        if (old) {
+                                old->m.mnt_opts = update_shared_subtree_options (
+                                        old->m.mnt_opts, flags);
+                                update_mtab (mnt.mnt_dir, &old->m);
+                        } else if (!mount_all) {
+                                error(_("mount: warning: couldn't update %s, because "
+                                        "no previous entry was found for %s"),
+                                      MOUNTED, mnt.mnt_dir);
+                        }
+                } else {
                        mntFILE *mfp;

+                        if (flags & (MS_BIND|MS_MOVE)) {
+                                old = getmntdirbackward(mnt.mnt_fsname, NULL);
+                                my_free(mnt.mnt_fsname);
+                                if (old) {
+                                        mnt.mnt_fsname =
+                                                canonicalize (old->m.mnt_fsname);
+                                        mnt.mnt_type = old->m.mnt_type;
+                                        mnt.mnt_opts = old->m.mnt_opts;
+                                        mnt.mnt_freq = old->m.mnt_freq;
+                                        mnt.mnt_passno = old->m.mnt_passno;
+                                        if (flags & MS_MOVE)
+                                                update_mtab (old->m.mnt_dir, NULL);
+                                } else {
+                                        mnt.mnt_fsname = xstrdup("none");
+                                        mnt.mnt_type = "none";
+                                }
+                        }
```

```
+
                        lock_mtab();
                        mfp = my_setmntent(MOUNTED, "a+");
                        if (mfp == NULL || mfp->mntent_fp == NULL) {
@@ -1414,6 +1505,14 @@
        { "move", 0, 0, 133 },
        { "guess-fstype", 1, 0, 134 },
        { "rbind", 0, 0, 135 },
+       { "make-unbindable", 0, 0, 136 },
+       { "make-runbindable", 0, 0, 137 },
+       { "make-private", 0, 0, 138 },
+       { "make-rprivate", 0, 0, 139 },
+       { "make-slave", 0, 0, 140 },
+       { "make-rslave", 0, 0, 141 },
+       { "make-shared", 0, 0, 142 },
+       { "make-rshared", 0, 0, 143 },
        { "internal-only", 0, 0, 'i' },
        { NULL, 0, 0, 0 }
 };
@@ -1591,6 +1690,30 @@
                case 135:
                        mounttype = (MS_BIND | MS_REC);
                        break;
+               case 136: /* make unbindable */
+                       mounttype = MS_UNBINDABLE;
+                       break;
+               case 137:
+                       mounttype = (MS_UNBINDABLE | MS_REC);
+                       break;
+               case 138: /* make private */
+                       mounttype = MS_PRIVATE;
+                       break;
+               case 139:
+                       mounttype = (MS_PRIVATE | MS_REC);
+                       break;
+               case 140: /* make slave */
+                       mounttype = MS_SLAVE;
+                       break;
+               case 141:
+                       mounttype = (MS_SLAVE | MS_REC);
+                       break;
+               case 142: /* make shared */
+                       mounttype = MS_SHARED;
+                       break;
+               case 143:
+                       mounttype = (MS_SHARED | MS_REC);
+                       break;
                case '?':
                default:
                        usage(stderr, EX_USAGE);
@@ -1647,6 +1770,11 @@
                /* mount [-nfrvw] [-o options] special | node */
                if (types != NULL)
                        usage(stderr, EX_USAGE);
+               if (mounttype & (MS_UNBINDABLE|MS_PRIVATE|MS_SLAVE|MS_SHARED)) {
+                       result = mount_one("", argv[0], "none", NULL, options, 0, 0);
+                       break;
+               }
+
                if (specseen) {
                        /* We know the device. Where shall we mount it? */
                        mc = (uuid ? getfsuuidspec(uuid)
```

Listing 4: mount.8 patch

```
--- mount.8.old 2007-04-17 12:01:21.000000000 +0200
```

```
+++ mount.8       2007-04-20 12:36:14.000000000 +0200
@@ -131,6 +131,29 @@
 .B "mount --move olddir newdir"
 .RE

+In Linux 2.6.15 a shared subtree system for mount points was
+introduced. This allows using the --bind and --rbind options to not
+just make copies, but actual clones that change along when new
+submounts are made.  A typical call would be
+.RS
+.br
+.B "mount --make-shared olddir"
+.br
+.B "mount --bind olddir newdir"
+.RE
+This sets up a bi-directional propagation of submounts: anything
+mounted under
+.IR olddir
+will show up under
+.IR newdir
+and vice versa.  It's also possible to set up uni-directional
+propagation (with --make-slave), to make a mount point unavailable for
+--bind/--rbind (with --make-unbindable), and to undo any of these
+(with --make-private).  For more information, see the section on
+shared subtrees below, or the
+.IR Documentation/sharedsubtree.txt
+file in the kernel source.
+
 The
 .I proc
 file system is not associated with a special device, and when
@@ -598,6 +621,9 @@
 .B nomand
 Do not allow mandatory locks on this filesystem.
 .TP
+.B norelatime
+Update inode access time for each access. This is the default.
+.TP
 .B nosuid
 Do not allow set-user-identifier or set-group-identifier bits to take
 effect. (This seems safe, but is in fact rather unsafe if you have
@@ -615,6 +641,10 @@
 (unless overridden by subsequent options, as in the option line
 .BR owner,dev,suid ).
 .TP
+.B relatime
+Update inode access time only when the access time is less than or
+equal to the modify or change time.
+.TP
 .B remount
 Attempt to remount an already-mounted file system.  This is commonly
 used to change the mount flags for a file system, especially to make a
@@ -655,12 +685,30 @@
 .BR users,exec,dev,suid ).
 .RE
 .TP
-.B \-\-bind
+.B \-\-bind/\-\-rbind
 Remount a subtree somewhere else (so that its contents are available
 in both places). See above.
 .TP
 .B \-\-move
 Move a subtree to some other place. See above.
+.TP
+.B \-\-make-shared/\-\-make-rshared
+Make a mount point shared, so that subsequent binds set up
```

```
+bi-directional propagation of submounts. See above.
+.TP
+.B \-\-make-slave/\-\-make-rslave
+Make a mount point that is shared the slave of the mount points it
+shares with.  This will make submounts (only) propagate from those
+masters to the slave, and not vice versa. See above.
+.TP
+.B \-\-make-unbindable/\-\-make-runbindable
+Make a mount point unable to be the source of subsequent binds. See
+above.
+.TP
+.B \-\-make-private/\-\-make-rprivate
+Mount points are in the private state by default. This option can make
+a mount point private again, after it was made shared, slave or
+unbindable. See above.

 .SH "FILESYSTEM SPECIFIC MOUNT OPTIONS"
 The following options apply only to certain file systems.
@@ -1863,6 +1911,128 @@
 You can also free a loop device by hand, using 'losetup -d', see
 .BR losetup (8).

+.SH "SHARED SUBTREES"
+The shared subtree system allows setting up propagation of mount and
+unmount events between mount points using --bind/--rbind.  There are 4
+states a mount point can be in, and its state affects the way a
+subsequent bind with this mount point as the source will work:
+
+.B Private. \fP(This is the default state.)
+.RS
+Private mount points act as per the pre-2.6.15 semantics: a bind is a
+simple copy and no new events will propagate between the bind's source
+and the destination afterwards.
+
+If a mount point
+.I /mnt/a
+is in another state, it can be made private again with:
+
+.nf
+.B "  mount --make-private /mnt/a"
+.fi
+.RE
+
+.B Shared.
+.RS
+When a shared mount point is bound to another mount point, subsequent
+mount and unmounts events under either will propagate to the other.
+
+For example, suppose that
+.IR /mnt/a
+names a mount point.  Then we could make it shared and bind it to
+.IR /mnt/b
+as follows:
+
+.nf
+.B "  mount --make-shared /mnt/a"
+.B "  mount --bind /mnt/a /mnt/b"
+.fi
+
+Both
+.IR /mnt/a
+and
+.IR /mnt/b
+are now in the shared state, and anything we (un)mount under either
+.IR /mnt/a
+or
```

```
+.IR /mnt/b
+will be (un)mounted under both:
+.IR /mnt/a
+and
+.IR /mnt/b
+form a ‘peer group’.
+.RE
+
+.B Slave.
+.RS
+A slave mount point is similar to a shared mount point, but events
+are only propagated towards it, not from it. A mount point can only
+be made a slave if it is part of a peer group first. It then
+becomes slave to all its former peers, receiving propagation events
+from them, but not sending its own to them.
+
+Continuing the example above, we could make
+.IR /mnt/a
+a slave to
+.IR /mnt/b
+as follows:
+
+.nf
+.B "  mount --make-slave /mnt/a"
+.fi
+
+Then, any (un)mounting under
+.IR /mnt/b
+would show up under
+.IR /mnt/a
+but not vice versa.
+
+Note that we could have just as well made
+.IR /mnt/b
+a slave to
+.IR /mnt/a
+instead, in the exact same fashion. That
+.IR /mnt/a
+was the source and
+.IR /mnt/b
+the destination of the bind that set up their peer group is of no
+relevance.
+.RE
+
+.B Unbindable.
+.RS
+An unbindable mount point is simply one that cannot be the source of a
+bind operation. Trying to do:
+
+.nf
+.B "  mount --make-unbindable /mnt/a"
+.B "  mount --bind /mnt/a /mnt/b"
+.fi
+
+will result in an error message for the bind.
+
+Note that both --make-private and --make-unbindable also sever the
+propagation links of the target mount point to and from any peers it
+has.
+.RE
+
+Apart from these 4 states there's actually (more or less) a fifth
+state. When a mount point is made a slave to its peers, it can then
+be made shared again with --make-shared. Such a mount point is said
+to be in the ‘shared and slave’ state.
+
```

```
+The mount point will still be slave to the peers it had at the time of
+the --make-slave, thus receiving propagation events from them, but not
+sending them any.  However, it can now also get new peers (through
+--bind) with which it will have two-way propagation.  (And by
+`transitivity of slavery' its new peers will receive events from its
+old peers, but not the other way around.)
+
+More details on the semantics of the shared subtrees system can be
+found in the kernel source documentation, in the file
+.IR Documentation/sharedsubtree.txt
+.
+
 .SH RETURN CODES
 .B mount
 has the following return codes (the bits can be ORed):
```