```
diff -Nur linux-2.4.18.orig/fs/namei.c linux-2.4.18/fs/namei.c
--- linux-2.4.18.orig/fs/namei.c        2004-04-15 21:22:48.000000000 +0800
+++ linux-2.4.18/fs/namei.c     2004-04-24 14:37:00.000000000 +0800
@@ -581,9 +581,9 @@
                        if (err < 0)
                                break;
                }
-               dentry = cached_lookup(nd->dentry, &this, 0);
+               dentry = cached_lookup(nd->dentry, &this, nd->flags);
                if (!dentry) {
-                       dentry = real_lookup(nd->dentry, &this, 0);
+                       dentry = real_lookup(nd->dentry, &this, nd->flags);
                        err = PTR_ERR(dentry);
                        if (IS_ERR(dentry))
                                break;
diff -Nur linux-2.4.18.orig/Documentation/Configure.help linux-2.4.18/Documentation/Configure.
--- linux-2.4.18.orig/Documentation/Configure.help      2004-04-15 21:24:21.000000000 +0800
+++ linux-2.4.18/Documentation/Configure.help   2004-04-24 14:38:31.000000000 +0800
@@ -14723,7 +14723,7 @@
   automounter (amd), which is a pure user space daemon.

   To use the automounter you need the user-space tools from
-  <ftp://ftp.kernel.org/pub/linux/daemons/autofs/testing-v4/>; you also
+  <ftp://ftp.kernel.org/pub/linux/daemons/autofs/v4/>; you also
   want to answer Y to "NFS file system support", below.

   If you want to compile this as a module ( = code which can be
diff -Nur linux-2.4.18/fs/autofs4/autofs_i.h autofs4-2.4/fs/autofs4/autofs_i.h
--- linux-2.4.18/fs/autofs4/autofs_i.h  2002-04-18 19:33:01.000000000 +0800
+++ autofs4-2.4/fs/autofs4/autofs_i.h   2004-05-08 12:45:20.000000000 +0800
@@ -80,7 +80,7 @@
        char *name;
        /* This is for status reporting upon return */
        int status;
-       int wait_ctr;
+       atomic_t wait_ctr;
 };

 #define AUTOFS_SBI_MAGIC 0x6d4a556d
@@ -91,7 +91,11 @@
        pid_t oz_pgrp;
        int catatonic;
        int version;
+       int sub_version;
        unsigned long exp_timeout;
+       int reghost_enabled;
+       int needs_reghost;
+       struct semaphore wq_sem;
        struct super_block *sb;
        struct autofs_wait_queue *queues; /* Wait queue pointer */
 };
@@ -123,6 +127,12 @@
                (inf != NULL && inf->flags & AUTOFS_INF_EXPIRING);
 }

+static inline void autofs4_copy_atime(struct file *src, struct file *dst)
+{
+       dst->f_dentry->d_inode->i_atime = src->f_dentry->d_inode->i_atime;
+       return;
+}
```

```
+
 struct inode *autofs4_get_inode(struct super_block *, struct autofs_info *);
 struct autofs_info *autofs4_init_inf(struct autofs_sb_info *, mode_t mode);
 void autofs4_free_ino(struct autofs_info *);
@@ -156,6 +166,56 @@
        NFY_EXPIRE
 };

-int autofs4_wait(struct autofs_sb_info *,struct qstr *, enum autofs_notify);
+int autofs4_wait(struct autofs_sb_info *,struct dentry *, enum autofs_notify);
 int autofs4_wait_release(struct autofs_sb_info *,autofs_wqt_t,int);
 void autofs4_catatonic_mode(struct autofs_sb_info *);
+
+#if !defined(REDHAT_KERNEL) || LINUX_VERSION_CODE < KERNEL_VERSION(2,4,19)
+/**
+ * list_for_each_entry  -       iterate over list of given type
+ * @pos:        the type * to use as a loop counter.
+ * @head:       the head for your list.
+ * @member:     the name of the list_struct within the struct.
+ */
+#define list_for_each_entry(pos, head, member)                          \
+        for (pos = list_entry((head)->next, typeof(*pos), member),      \
+                     prefetch(pos->member.next);                        \
+             &pos->member != (head);                                    \
+             pos = list_entry(pos->member.next, typeof(*pos), member),  \
+                     prefetch(pos->member.next))
+
+#endif
+
+static inline int simple_positive(struct dentry *dentry)
+{
+       return dentry->d_inode && !d_unhashed(dentry);
+}
+
+static inline int simple_empty_nolock(struct dentry *dentry)
+{
+       struct dentry *child;
+       int ret = 0;
+
+       list_for_each_entry(child, &dentry->d_subdirs, d_child)
+               if (simple_positive(child))
+                       goto out;
+       ret = 1;
+out:
+       return ret;
+}
+
+static inline int simple_empty(struct dentry *dentry)
+{
+       struct dentry *child;
+       int ret = 0;
+
+       spin_lock(&dcache_lock);
+       list_for_each_entry(child, &dentry->d_subdirs, d_child)
+               if (simple_positive(child))
+                       goto out;
+       ret = 1;
+out:
+       spin_unlock(&dcache_lock);
+       return ret;
```

```
+}
+
diff -Nur linux-2.4.18/fs/autofs4/expire.c autofs4-2.4/fs/autofs4/expire.c
--- linux-2.4.18/fs/autofs4/expire.c    2001-06-12 10:15:27.000000000 +0800
+++ autofs4-2.4/fs/autofs4/expire.c     2004-05-08 12:45:20.000000000 +0800
@@ -4,6 +4,7 @@
  *
  *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
  *  Copyright 1999-2000 Jeremy Fitzhardinge <jeremy@goop.org>
+ *  Copyright 2001-2003 Ian Kent <raven@themaw.net>
  *
  * This file is part of the Linux kernel and is made available under
  * the terms of the GNU General Public License, version 2, or at your
@@ -13,134 +14,238 @@

 #include "autofs_i.h"

-/*
- * Determine if a subtree of the namespace is busy.
- *
- * mnt is the mount tree under the autofs mountpoint
- */
-static inline int is_vfsmnt_tree_busy(struct vfsmount *mnt)
+static unsigned long now;
+
+/* Check if a dentry can be expired return 1 if it can else return 0 */
+static inline int autofs4_can_expire(struct dentry *dentry,
+                                     unsigned long timeout, int do_now)
+{
+       struct autofs_info *ino = autofs4_dentry_ino(dentry);
+
+       /* dentry in the process of being deleted */
+       if (ino == NULL)
+               return 0;
+
+       /* No point expiring a pending mount */
+       if (dentry->d_flags & DCACHE_AUTOFS_PENDING)
+               return 0;
+
+       if (!do_now) {
+               /* Too young to die */
+               if (time_after(ino->last_used + timeout, now))
+                       return 0;
+
+               /* update last_used here :-
+                  - obviously makes sense if it is in use now
+                  - less obviously, prevents rapid-fire expire
+                    attempts if expire fails the first time */
+               ino->last_used = now;
+       }
+
+       return 1;
+}
+
+/* Sorry I can't solve the problem without using counts either */
+static int autofs4_may_umount(struct vfsmount *mnt)
 {
-       struct vfsmount *this_parent = mnt;
        struct list_head *next;
-       int count;
```

```
+       struct vfsmount *this_parent = mnt;
+       int actual_refs;
+       int minimum_refs;

-       count = atomic_read(&mnt->mnt_count) - 1;
+       actual_refs = atomic_read(&mnt->mnt_count);
+       minimum_refs = 2;

+       spin_lock(&dcache_lock);
 repeat:
        next = this_parent->mnt_mounts.next;
-       DPRINTK(("is_vfsmnt_tree_busy: mnt=%p, this_parent=%p, next=%p\n",
-                mnt, this_parent, next));
 resume:
-       for( ; next != &this_parent->mnt_mounts; next = next->next) {
-               struct vfsmount *p = list_entry(next, struct vfsmount,
-                                               mnt_child);
-
-               /* -1 for struct vfs_mount's normal count,
-                  -1 to compensate for child's reference to parent */
-               count += atomic_read(&p->mnt_count) - 1 - 1;
+       while (next != &this_parent->mnt_mounts) {
+               struct vfsmount *p = list_entry(next, struct vfsmount, mnt_child);
+
+               next = next->next;

-               DPRINTK(("is_vfsmnt_tree_busy: p=%p, count now %d\n",
-                        p, count));
+               actual_refs += atomic_read(&p->mnt_count);
+               minimum_refs += 2;

                if (!list_empty(&p->mnt_mounts)) {
                        this_parent = p;
                        goto repeat;
                }
-               /* root is busy if any leaf is busy */
-               if (atomic_read(&p->mnt_count) > 1)
-                       return 1;
        }

-       /* All done at this level ... ascend and resume the search. */
        if (this_parent != mnt) {
-               next = this_parent->mnt_child.next;
+               next = this_parent->mnt_child.next;
                this_parent = this_parent->mnt_parent;
                goto resume;
        }
+       spin_unlock(&dcache_lock);
+
+       DPRINTK(("autofs4_may_umount: done actual = %d, minimum = %d\n",
+                actual_refs, minimum_refs));

-       DPRINTK(("is_vfsmnt_tree_busy: count=%d\n", count));
-       return count != 0; /* remaining users? */
+       return actual_refs > minimum_refs;
 }

-/* Traverse a dentry's list of vfsmounts and return the number of
-   non-busy mounts */
-static int check_vfsmnt(struct vfsmount *mnt, struct dentry *dentry)
```

```
+/* Check a mount point for busyness return 1 if not busy, otherwise */
+static int autofs4_check_mount(struct vfsmount *mnt, struct dentry *dentry)
 {
-       int ret = dentry->d_mounted;
-       struct vfsmount *vfs = lookup_mnt(mnt, dentry);
+       int status = 0;

-       if (vfs && is_vfsmnt_tree_busy(vfs))
-               ret--;
-       DPRINTK(("check_vfsmnt: ret=%d\n", ret));
-       return ret;
+       DPRINTK(("autofs4_check_mount: dentry %p %.*s\n",
+                dentry, (int)dentry->d_name.len, dentry->d_name.name));
+
+       mntget(mnt);
+       dget(dentry);
+
+       if (!follow_down(&mnt, &dentry))
+               goto done;
+
+       while (d_mountpoint(dentry) && follow_down(&mnt, &dentry))
+               ;
+
+       /* This is an autofs submount, we can't expire it */
+       if (is_autofs4_dentry(dentry))
+               goto done;
+
+       /* The big question */
+       if (autofs4_may_umount(mnt) == 0)
+               status = 1;
+done:
+       DPRINTK(("autofs4_check_mount: returning = %d\n", status));
+       mntput(mnt);
+       dput(dentry);
+       return status;
 }

-/* Check dentry tree for busyness.  If a dentry appears to be busy
-   because it is a mountpoint, check to see if the mounted
-   filesystem is busy. */
-static int is_tree_busy(struct vfsmount *topmnt, struct dentry *top)
+/* Check a directory tree of mount points for busyness
+ * The tree is not busy iff no mountpoints are busy
+ * Return 1 if the tree is busy or 0 otherwise
+ */
+static int autofs4_check_tree(struct vfsmount *mnt,
+                             struct dentry *top,
+                             unsigned long timeout,
+                             int do_now)
 {
-       struct dentry *this_parent;
+       struct dentry *this_parent = top;
        struct list_head *next;
-       int count;

-       count = atomic_read(&top->d_count);
-
-       DPRINTK(("is_tree_busy: top=%p initial count=%d\n",
-                top, count));
-       this_parent = top;
```

```
-
-       if (is_autofs4_dentry(top)) {
-               count--;
-               DPRINTK(("is_tree_busy: autofs; count=%d\n", count));
-       }
+       DPRINTK(("autofs4_check_tree: parent %p %.*s\n",
+                top, (int)top->d_name.len, top->d_name.name));

-       if (d_mountpoint(top))
-               count -= check_vfsmnt(topmnt, top);
+       /* Negative dentry - give up */
+       if (!simple_positive(top))
+               return 0;
+
+       /* Timeout of a tree mount is determined by its top dentry */
+       if (!autofs4_can_expire(top, timeout, do_now))
+               return 0;

- repeat:
+       spin_lock(&dcache_lock);
+repeat:
        next = this_parent->d_subdirs.next;
- resume:
+resume:
        while (next != &this_parent->d_subdirs) {
-               int adj = 0;
-               struct dentry *dentry = list_entry(next, struct dentry,
-                                                  d_child);
+               struct dentry *dentry = list_entry(next, struct dentry, d_child);
+
+               /* Negative dentry - give up */
+               if (!simple_positive(dentry)) {
+                       next = next->next;
+                       continue;
+               }
+
+               DPRINTK(("autofs4_check_tree: dentry %p %.*s\n",
+                        dentry, (int)dentry->d_name.len, dentry->d_name.name));
+
+               if (!list_empty(&dentry->d_subdirs)) {
+                       this_parent = dentry;
+                       goto repeat;
+               }
+
+               dentry = dget(dentry);
+               spin_unlock(&dcache_lock);
+
+               if (d_mountpoint(dentry)) {
+                       /* First busy => tree busy */
+                       if (!autofs4_check_mount(mnt, dentry)) {
+                               dput(dentry);
+                               return 0;
+                       }
+               }
+
+               dput(dentry);
+               spin_lock(&dcache_lock);
                next = next->next;
+       }
+
```

```
+       if (this_parent != top) {
+               next = this_parent->d_child.next;
+               this_parent = this_parent->d_parent;
+               goto resume;
+       }
+       spin_unlock(&dcache_lock);
+
+       return 1;
+}
+
+struct dentry *autofs4_check_leaves(struct vfsmount *mnt,
+                                   struct dentry *parent,
+                                   unsigned long timeout,
+                                   int do_now)
+{
+       struct dentry *this_parent = parent;
+       struct list_head *next;

-               count += atomic_read(&dentry->d_count) - 1;
+       DPRINTK(("autofs4_check_leaves: parent %p %.*s\n",
+                parent, (int)parent->d_name.len, parent->d_name.name));

-               if (d_mountpoint(dentry))
-                       adj += check_vfsmnt(topmnt, dentry);
+       spin_lock(&dcache_lock);
+repeat:
+       next = this_parent->d_subdirs.next;
+resume:
+       while (next != &this_parent->d_subdirs) {
+               struct dentry *dentry = list_entry(next, struct dentry, d_child);

-               if (is_autofs4_dentry(dentry)) {
-                       adj++;
-                       DPRINTK(("is_tree_busy: autofs; adj=%d\n",
-                                adj));
+               /* Negative dentry - give up */
+               if (!simple_positive(dentry)) {
+                       next = next->next;
+                       continue;
                }

-               count -= adj;
+               DPRINTK(("autofs4_check_leaves: dentry %p %.*s\n",
+                        dentry, (int)dentry->d_name.len, dentry->d_name.name));

                if (!list_empty(&dentry->d_subdirs)) {
                        this_parent = dentry;
                        goto repeat;
                }

-               if (atomic_read(&dentry->d_count) != adj) {
-                       DPRINTK(("is_tree_busy: busy leaf (d_count=%d adj=%d)\n",
-                                atomic_read(&dentry->d_count), adj));
-                       return 1;
+               dentry = dget(dentry);
+               spin_unlock(&dcache_lock);
+
+               if (d_mountpoint(dentry)) {
+                       /* Can we expire this guy */
+                       if (!autofs4_can_expire(dentry, timeout, do_now))
```

```
+                               goto cont;
+
+                       /* Can we umount this guy */
+                       if (autofs4_check_mount(mnt, dentry))
+                               return dentry;
                }
+cont:
+               dput(dentry);
+               spin_lock(&dcache_lock);
+               next = next->next;
        }

-       /* All done at this level ... ascend and resume the search. */
-       if (this_parent != top) {
-               next = this_parent->d_child.next;
+       if (this_parent != parent) {
+               next = this_parent->d_child.next;
                this_parent = this_parent->d_parent;
                goto resume;
        }
+       spin_unlock(&dcache_lock);

-       DPRINTK(("is_tree_busy: count=%d\n", count));
-       return count != 0; /* remaining users? */
+       return NULL;
 }

 /*
@@ -152,61 +257,86 @@
 static struct dentry *autofs4_expire(struct super_block *sb,
                                     struct vfsmount *mnt,
                                     struct autofs_sb_info *sbi,
-                                    int do_now)
+                                    int how)
 {
-       unsigned long now = jiffies;
        unsigned long timeout;
        struct dentry *root = sb->s_root;
-       struct list_head *tmp;
+       struct dentry *expired = NULL;
+       struct list_head *next;
+       int do_now = how & AUTOFS_EXP_IMMEDIATE;
+       int exp_leaves = how & AUTOFS_EXP_LEAVES;

        if (!sbi->exp_timeout || !root)
                return NULL;

+       now = jiffies;
        timeout = sbi->exp_timeout;

        spin_lock(&dcache_lock);
-       for(tmp = root->d_subdirs.next;
-           tmp != &root->d_subdirs;
-           tmp = tmp->next) {
-               struct autofs_info *ino;
-               struct dentry *dentry = list_entry(tmp, struct dentry, d_child);
+       next = root->d_subdirs.next;

-               if (dentry->d_inode == NULL)
+       /* On exit from the loop expire is set to a dgot dentry
```

```
+        * to expire or it's NULL */
+       while (next != &root->d_subdirs) {
+               struct dentry *dentry = list_entry(next, struct dentry, d_child);
+
+               /* Negative dentry - give up */
+               if (!simple_positive(dentry)) {
+                       next = next->next;
                        continue;
+               }

-               ino = autofs4_dentry_ino(dentry);
+               dentry = dget(dentry);
+               spin_unlock(&dcache_lock);

-               if (ino == NULL) {
-                       /* dentry in the process of being deleted */
-                       continue;
+               /* Case 1: indirect mount or top level direct mount */
+               if (d_mountpoint(dentry)) {
+                       DPRINTK(("autofs4_expire: checking mountpoint %p %.*s\n",
+                        dentry, (int)dentry->d_name.len, dentry->d_name.name));
+
+                       /* Can we expire this guy */
+                       if (!autofs4_can_expire(dentry, timeout, do_now))
+                               goto next;
+
+                       /* Can we umount this guy */
+                       if (autofs4_check_mount(mnt, dentry)) {
+                               expired = dentry;
+                               break;
+                       }
+                       goto next;
                }

-               /* No point expiring a pending mount */
-               if (dentry->d_flags & DCACHE_AUTOFS_PENDING)
-                       continue;
+               if (simple_empty(dentry))
+                       goto next;

-               if (!do_now) {
-                       /* Too young to die */
-                       if (time_after(ino->last_used + timeout, now))
-                               continue;
-
-                       /* update last_used here :-
-                          - obviously makes sense if it is in use now
-                          - less obviously, prevents rapid-fire expire
-                            attempts if expire fails the first time */
-                       ino->last_used = now;
+               /* Case 2: tree mount, expire iff entire tree is not busy */
+               if (!exp_leaves) {
+                       if (autofs4_check_tree(mnt, dentry, timeout, do_now)) {
+                               expired = dentry;
+                               break;
+                       }
+               /* Case 3: direct mount, expire individual leaves */
+               } else {
+                       expired = autofs4_check_leaves(mnt, dentry, timeout, do_now);
+                       if (expired) {
```

```
+                               dput(dentry);
+                               break;
+                       }
                }
-               if (!is_tree_busy(mnt, dentry)) {
-                       DPRINTK(("autofs_expire: returning %p %.*s\n",
-                                dentry, (int)dentry->d_name.len, dentry->d_name.name));
-                       /* Start from here next time */
-                       list_del(&root->d_subdirs);
-                       list_add(&root->d_subdirs, &dentry->d_child);
-                       dget(dentry);
-                       spin_unlock(&dcache_lock);
+next:
+               dput(dentry);
+               spin_lock(&dcache_lock);
+               next = next->next;
+       }

-                       return dentry;
-               }
+       if ( expired ) {
+               DPRINTK(("autofs4_expire: returning %p %.*s\n",
+                        expired, (int)expired->d_name.len, expired->d_name.name));
+               spin_lock(&dcache_lock);
+               list_del(&expired->d_parent->d_subdirs);
+               list_add(&expired->d_parent->d_subdirs, &expired->d_child);
+               spin_unlock(&dcache_lock);
+               return expired;
        }
        spin_unlock(&dcache_lock);

@@ -259,11 +389,11 @@
                /* This is synchronous because it makes the daemon a
                    little easier */
                de_info->flags |= AUTOFS_INF_EXPIRING;
-               ret = autofs4_wait(sbi, &dentry->d_name, NFY_EXPIRE);
+               ret = autofs4_wait(sbi, dentry, NFY_EXPIRE);
                de_info->flags &= ~AUTOFS_INF_EXPIRING;
                dput(dentry);
        }
-
+
        return ret;
 }

diff -Nur linux-2.4.18/fs/autofs4/inode.c autofs4-2.4/fs/autofs4/inode.c
--- linux-2.4.18/fs/autofs4/inode.c     2001-11-10 06:11:14.000000000 +0800
+++ autofs4-2.4/fs/autofs4/inode.c      2004-05-08 12:45:20.000000000 +0800
@@ -87,7 +87,7 @@

        kfree(sbi);

-       DPRINTK(("autofs: shutting down\n"));
+       DPRINTK(("autofs4: shutting down\n"));
 }

 static int autofs4_statfs(struct super_block *sb, struct statfs *buf);
@@ -181,12 +181,13 @@
        struct file * pipe;
        int pipefd;
```

```
        struct autofs_sb_info *sbi;
+       struct autofs_info *ino;
        int minproto, maxproto;

        sbi = (struct autofs_sb_info *) kmalloc(sizeof(*sbi), GFP_KERNEL);
        if ( !sbi )
                goto fail_unlock;
-       DPRINTK(("autofs: starting up, sbi = %p\n",sbi));
+       DPRINTK(("autofs4: starting up, sbi = %p\n",sbi));

        memset(sbi, 0, sizeof(*sbi));

@@ -197,7 +198,11 @@
        sbi->oz_pgrp = current->pgrp;
        sbi->sb = s;
        sbi->version = 0;
+       sbi->sub_version = 0;
+       init_MUTEX(&sbi->wq_sem);
        sbi->queues = NULL;
+       sbi->reghost_enabled = 0;
+       sbi->needs_reghost = 0;
        s->s_blocksize = 1024;
        s->s_blocksize_bits = 10;
        s->s_magic = AUTOFS_SUPER_MAGIC;
@@ -206,7 +211,12 @@
        /*
         * Get the root inode and dentry, but defer checking for errors.
         */
-       root_inode = autofs4_get_inode(s, autofs4_mkroot(sbi));
+       ino = autofs4_mkroot(sbi);
+       root_inode = autofs4_get_inode(s, ino);
+       kfree(ino);
+       if (!root_inode)
+               goto fail_free;
+
        root_inode->i_op = &autofs4_root_inode_operations;
        root_inode->i_fop = &autofs4_root_operations;
        root = d_alloc_root(root_inode);
@@ -220,14 +230,14 @@
                        &root_inode->i_uid, &root_inode->i_gid,
                        &sbi->oz_pgrp,
                        &minproto, &maxproto)) {
-               printk("autofs: called with bogus options\n");
+               printk("autofs4: called with bogus options\n");
                goto fail_dput;
        }

        /* Couldn't this be tested earlier? */
        if (maxproto < AUTOFS_MIN_PROTO_VERSION ||
            minproto > AUTOFS_MAX_PROTO_VERSION) {
-               printk("autofs: kernel does not match daemon version "
+               printk("autofs4: kernel does not match daemon version "
                       "daemon (%d, %d) kernel (%d, %d)\n",
                        minproto, maxproto,
                        AUTOFS_MIN_PROTO_VERSION, AUTOFS_MAX_PROTO_VERSION);
@@ -235,12 +245,13 @@
        }

        sbi->version = maxproto > AUTOFS_MAX_PROTO_VERSION ? AUTOFS_MAX_PROTO_VERSION : maxpro
+       sbi->sub_version =  AUTOFS_PROTO_SUBVERSION;
```

```
-       DPRINTK(("autofs: pipe fd = %d, pgrp = %u\n", pipefd, sbi->oz_pgrp));
+       DPRINTK(("autofs4: pipe fd = %d, pgrp = %u\n", pipefd, sbi->oz_pgrp));
        pipe = fget(pipefd);

        if ( !pipe ) {
-               printk("autofs: could not open pipe file descriptor\n");
+               printk("autofs4: could not open pipe file descriptor\n");
                goto fail_dput;
        }
        if ( !pipe->f_op || !pipe->f_op->write )
@@ -257,7 +268,7 @@
         * Failure ... clean up.
         */
 fail_fput:
-       printk("autofs: pipe file descriptor does not contain proper ops\n");
+       printk("autofs4: pipe file descriptor does not contain proper ops\n");
        /*
         * fput() can block, so we clear the super block first.
         */
@@ -270,7 +281,7 @@
        dput(root);
        goto fail_free;
 fail_iput:
-       printk("autofs: get root dentry failed\n");
+       printk("autofs4: get root dentry failed\n");
        /*
         * iput() can block, so we clear the super block first.
         */
diff -Nur linux-2.4.18/fs/autofs4/root.c autofs4-2.4/fs/autofs4/root.c
--- linux-2.4.18/fs/autofs4/root.c      2000-10-24 12:57:38.000000000 +0800
+++ autofs4-2.4/fs/autofs4/root.c       2004-05-08 12:45:20.000000000 +0800
@@ -4,6 +4,7 @@
  *
  *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
  *  Copyright 1999-2000 Jeremy Fitzhardinge <jeremy@goop.org>
+ *  Copyright 2001-2003 Ian Kent <raven@themaw.net>
  *
  * This file is part of the Linux kernel and is made available under
  * the terms of the GNU General Public License, version 2, or at your
@@ -16,6 +17,10 @@
 #include <linux/param.h>
 #include <linux/sched.h>
 #include <linux/smp_lock.h>
+#include <linux/file.h>
+#include <linux/limits.h>
+#include <linux/iobuf.h>
+#include <linux/module.h>
 #include "autofs_i.h"

 static struct dentry *autofs4_dir_lookup(struct inode *,struct dentry *);
@@ -24,17 +29,24 @@
 static int autofs4_dir_rmdir(struct inode *,struct dentry *);
 static int autofs4_dir_mkdir(struct inode *,struct dentry *,int);
 static int autofs4_root_ioctl(struct inode *, struct file *,unsigned int,unsigned long);
+static int autofs4_dir_open(struct inode *inode, struct file *file);
+static int autofs4_dir_close(struct inode *inode, struct file *file);
+static int autofs4_dir_readdir(struct file * filp, void * dirent, filldir_t filldir);
+static int autofs4_root_readdir(struct file * filp, void * dirent, filldir_t filldir);
 static struct dentry *autofs4_root_lookup(struct inode *,struct dentry *);
```

```
+static int autofs4_dcache_readdir(struct file *, void *, filldir_t);

 struct file_operations autofs4_root_operations = {
        read:           generic_read_dir,
-       readdir:        dcache_readdir,
+       readdir:        autofs4_root_readdir,
        ioctl:          autofs4_root_ioctl,
 };

 struct file_operations autofs4_dir_operations = {
+       open:           autofs4_dir_open,
+       release:        autofs4_dir_close,
        read:           generic_read_dir,
-       readdir:        dcache_readdir,
+       readdir:        autofs4_dir_readdir,
 };

 struct inode_operations autofs4_root_inode_operations = {
@@ -55,10 +67,11 @@

 /* Update usage from here to top of tree, so that scan of
    top-level directories will give a useful result */
-static void autofs4_update_usage(struct dentry *dentry)
+static inline void autofs4_update_usage(struct dentry *dentry)
 {
        struct dentry *top = dentry->d_sb->s_root;

+       spin_lock(&dcache_lock);
        for(; dentry != top; dentry = dentry->d_parent) {
                struct autofs_info *ino = autofs4_dentry_ino(dentry);

@@ -67,11 +80,213 @@
                        ino->last_used = jiffies;
                }
        }
+       spin_unlock(&dcache_lock);
+}
+
+static int autofs4_root_readdir(struct file *file, void *dirent, filldir_t filldir)
+{
+       struct autofs_sb_info *sbi = autofs4_sbi(file->f_dentry->d_sb);
+       int oz_mode = autofs4_oz_mode(sbi);
+
+       DPRINTK(("autofs4_root_readdir called, filp->f_pos = %lld\n", file->f_pos));
+
+       /*
+        * Don't set reghost flag if:
+        * 1) f_pos is larger than zero -- we've already been here.
+        * 2) we haven't even enabled reghosting in the 1st place.
+        * 3) this is the daemon doing a readdir
+        */
+       if ( oz_mode && file->f_pos == 0 && sbi->reghost_enabled )
+               sbi->needs_reghost = 1;
+
+       DPRINTK(("autofs4_root_readdir: needs_reghost = %d\n", sbi->needs_reghost));
+
+       return autofs4_dcache_readdir(file, dirent, filldir);
+}
+
+/*
```

```
+ * From 2.4 kernel readdir.c
+ */
+static int autofs4_dcache_readdir(struct file * filp, void * dirent, filldir_t filldir)
+{
+       int i;
+       struct dentry *dentry = filp->f_dentry;
+
+       i = filp->f_pos;
+       switch (i) {
+               case 0:
+                       if (filldir(dirent, ".", 1, i, dentry->d_inode->i_ino, DT_DIR) < 0)
+                               break;
+                       i++;
+                       filp->f_pos++;
+                       /* fallthrough */
+               case 1:
+                       if (filldir(dirent, "..", 2, i, dentry->d_parent->d_inode->i_ino, DT_D
+                               break;
+                       i++;
+                       filp->f_pos++;
+                       /* fallthrough */
+               default: {
+                       struct list_head *list;
+                       int j = i-2;
+
+                       spin_lock(&dcache_lock);
+                       list = dentry->d_subdirs.next;
+
+                       for (;;) {
+                               if (list == &dentry->d_subdirs) {
+                                       spin_unlock(&dcache_lock);
+                                       return 0;
+                               }
+                               if (!j)
+                                       break;
+                               j--;
+                               list = list->next;
+                       }
+
+                       while(1) {
+                               struct dentry *de = list_entry(list, struct dentry, d_child);
+
+                               if (!list_empty(&de->d_hash) && de->d_inode) {
+                                       spin_unlock(&dcache_lock);
+                                       if (filldir(dirent, de->d_name.name, de->d_name.len, f
+                                               break;
+                                       spin_lock(&dcache_lock);
+                               }
+                               filp->f_pos++;
+                               list = list->next;
+                               if (list != &dentry->d_subdirs)
+                                       continue;
+                               spin_unlock(&dcache_lock);
+                               break;
+                       }
+               }
+       }
+       return 0;
+}
+
```

```
+static int autofs4_dir_open(struct inode *inode, struct file *file)
+{
+	struct dentry *dentry = file->f_dentry;
+	struct vfsmount *mnt = file->f_vfsmnt;
+	struct autofs_sb_info *sbi = autofs4_sbi(dentry->d_sb);
+	int status;
+
+	DPRINTK(("autofs4_dir_open: file=%p dentry=%p %.*s\n",
+		 file, dentry, dentry->d_name.len, dentry->d_name.name));
+
+	if (autofs4_oz_mode(sbi))
+		goto out;
+
+	if (autofs4_ispending(dentry)) {
+		DPRINTK(("autofs4_dir_open: dentry busy\n"));
+		return -EBUSY;
+	}
+
+	if (!d_mountpoint(dentry) && dentry->d_op && dentry->d_op->d_revalidate) {
+		int empty;
+
+		/* In case there are stale directory dentrys from a failed mount */
+		spin_lock(&dcache_lock);
+		empty = list_empty(&dentry->d_subdirs);
+		spin_unlock(&dcache_lock);
+
+		if (!empty)
+			d_invalidate(dentry);
+
+		status = (dentry->d_op->d_revalidate)(dentry, LOOKUP_CONTINUE);
+
+		if ( !status )
+			return -ENOENT;
+	}
+
+	if ( d_mountpoint(dentry) ) {
+		struct file *fp = NULL;
+		struct vfsmount *fp_mnt = mntget(mnt);
+		struct dentry *fp_dentry = dget(dentry);
+
+		while (follow_down(&fp_mnt, &fp_dentry) && d_mountpoint(fp_dentry));
+
+		fp = dentry_open(fp_dentry, fp_mnt, file->f_flags);
+		status = PTR_ERR(fp);
+		if ( IS_ERR(fp) ) {
+			file->private_data = NULL;
+			return status;
+		}
+		file->private_data = fp;
+	}
+out:
+	return 0;
+}
+
+static int autofs4_dir_close(struct inode *inode, struct file *file)
+{
+	struct dentry *dentry = file->f_dentry;
+	struct autofs_sb_info *sbi = autofs4_sbi(dentry->d_sb);
+
+	DPRINTK(("autofs4_dir_close: file=%p dentry=%p %.*s\n",
```

```
+                file, dentry, dentry->d_name.len, dentry->d_name.name));
+
+       if ( autofs4_oz_mode(sbi) )
+               goto out;
+
+       if ( autofs4_ispending(dentry) ) {
+               DPRINTK(("autofs4_dir_close: dentry busy\n"));
+               return -EBUSY;
+       }
+
+       if ( d_mountpoint(dentry) ) {
+               struct file *fp = file->private_data;
+
+               if (!fp)
+                       return -ENOENT;
+
+               filp_close(fp, current->files);
+               file->private_data = NULL;
+       }
+out:
+       return 0;
+}
+
+static int autofs4_dir_readdir(struct file *file, void *dirent, filldir_t filldir)
+{
+       struct dentry *dentry = file->f_dentry;
+       struct autofs_sb_info *sbi = autofs4_sbi(dentry->d_sb);
+       int status;
+
+       DPRINTK(("autofs4_readdir: file=%p dentry=%p %.*s\n",
+                file, dentry, dentry->d_name.len, dentry->d_name.name));
+
+       if ( autofs4_oz_mode(sbi) )
+               goto out;
+
+       if ( autofs4_ispending(dentry) ) {
+               DPRINTK(("autofs4_readdir: dentry busy\n"));
+               return -EBUSY;
+       }
+
+       if ( d_mountpoint(dentry) ) {
+               struct file *fp = file->private_data;
+
+               if (!fp)
+                       return -ENOENT;
+
+               if (!fp->f_op || !fp->f_op->readdir)
+                       goto out;
+
+               status = vfs_readdir(fp, filldir, dirent);
+               file->f_pos = fp->f_pos;
+               if ( status )
+                       autofs4_copy_atime(file, fp);
+               return status;
+       }
+out:
+       return autofs4_dcache_readdir(file, dirent, filldir);
 }

 static int try_to_fill_dentry(struct dentry *dentry,
```

```
                               struct super_block *sb,
-                              struct autofs_sb_info *sbi)
+                              struct autofs_sb_info *sbi, int flags)
 {
        struct autofs_info *de_info = autofs4_dentry_ino(dentry);
        int status = 0;
@@ -80,12 +295,11 @@
            when expiration is done to trigger mount request with a new
            dentry */
        if (de_info && (de_info->flags & AUTOFS_INF_EXPIRING)) {
-               DPRINTK(("try_to_fill_entry: waiting for expire %p name=%.*s, flags&PENDING=%s
-                        dentry, dentry->d_name.len, dentry->d_name.name,
-                        dentry->d_flags & DCACHE_AUTOFS_PENDING?"t":"f",
-                        de_info, de_info?de_info->flags:0));
-               status = autofs4_wait(sbi, &dentry->d_name, NFY_NONE);
-
+               DPRINTK(("try_to_fill_entry: waiting for expire %p name=%.*s\n",
+                        dentry, dentry->d_name.len, dentry->d_name.name));
+
+               status = autofs4_wait(sbi, dentry, NFY_NONE);
+
                DPRINTK(("try_to_fill_entry: expire done status=%d\n", status));

                return 0;
@@ -95,12 +309,12 @@
                 dentry, dentry->d_name.len, dentry->d_name.name, dentry->d_inode));

        /* Wait for a pending mount, triggering one if there isn't one already */
-       while(dentry->d_inode == NULL) {
-               DPRINTK(("try_to_fill_entry: waiting for mount name=%.*s, de_info=%p de_info->
-                        dentry->d_name.len, dentry->d_name.name,
-                        de_info, de_info?de_info->flags:0));
-               status = autofs4_wait(sbi, &dentry->d_name, NFY_MOUNT);
-
+       if (dentry->d_inode == NULL) {
+               DPRINTK(("try_to_fill_entry: waiting for mount name=%.*s\n",
+                        dentry->d_name.len, dentry->d_name.name));
+
+               status = autofs4_wait(sbi, dentry, NFY_MOUNT);
+
                DPRINTK(("try_to_fill_entry: mount done status=%d\n", status));

                if (status && dentry->d_inode)
@@ -115,19 +329,22 @@
                        /* Return a negative dentry, but leave it "pending" */
                        return 1;
                }
-       }
+       /* Trigger mount for path component or follow link */
+       } else if ( flags & (LOOKUP_CONTINUE | LOOKUP_DIRECTORY) ||
+                       current->link_count ) {
+               DPRINTK(("try_to_fill_entry: waiting for mount name=%.*s\n",
+                        dentry->d_name.len, dentry->d_name.name));

-       /* If this is an unused directory that isn't a mount point,
-          bitch at the daemon and fix it in user space */
-       spin_lock(&dcache_lock);
-       if (S_ISDIR(dentry->d_inode->i_mode) &&
-           !d_mountpoint(dentry) &&
-           list_empty(&dentry->d_subdirs)) {
```

```
-               DPRINTK(("try_to_fill_entry: mounting existing dir\n"));
-               spin_unlock(&dcache_lock);
-               return autofs4_wait(sbi, &dentry->d_name, NFY_MOUNT) == 0;
+               dentry->d_flags |= DCACHE_AUTOFS_PENDING;
+               status = autofs4_wait(sbi, dentry, NFY_MOUNT);
+
+               DPRINTK(("try_to_fill_entry: mount done status=%d\n", status));
+
+               if ( status ) {
+                       dentry->d_flags &= ~DCACHE_AUTOFS_PENDING;
+                       return 0;
+               }
        }
-       spin_unlock(&dcache_lock);

        /* We don't update the usages for the autofs daemon itself, this
           is necessary for recursive autofs mounts */
@@ -138,46 +355,46 @@
        return 1;
 }

-
 /*
  * Revalidate is called on every cache lookup.  Some of those
  * cache lookups may actually happen while the dentry is not
  * yet completely filled in, and revalidate has to delay such
  * lookups..
  */
-static int autofs4_root_revalidate(struct dentry * dentry, int flags)
+static int autofs4_revalidate(struct dentry * dentry, int flags)
 {
        struct inode * dir = dentry->d_parent->d_inode;
        struct autofs_sb_info *sbi = autofs4_sbi(dir->i_sb);
-       struct autofs_info *ino;
        int oz_mode = autofs4_oz_mode(sbi);
+       int status = 1;
+
+       DPRINTK(("autofs4_revalidate: dentry=%p %.*s flags=%d\n",
+                dentry, dentry->d_name.len, dentry->d_name.name, flags));

        /* Pending dentry */
        if (autofs4_ispending(dentry)) {
-               if (autofs4_oz_mode(sbi))
-                       return 1;
-               else
-                       return try_to_fill_dentry(dentry, dir->i_sb, sbi);
+               if ( !oz_mode )
+                       status = try_to_fill_dentry(dentry, dir->i_sb, sbi, flags);
+               return status;
        }

        /* Negative dentry.. invalidate if "old" */
-       if (dentry->d_inode == NULL)
-               return (dentry->d_time - jiffies <= AUTOFS_NEGATIVE_TIMEOUT);
-
-       ino = autofs4_dentry_ino(dentry);
+       if (dentry->d_inode == NULL) {
+               status = (dentry->d_time - jiffies <= AUTOFS_NEGATIVE_TIMEOUT);
+               return status;
+       }
```

```
        /* Check for a non-mountpoint directory with no contents */
        spin_lock(&dcache_lock);
        if (S_ISDIR(dentry->d_inode->i_mode) &&
            !d_mountpoint(dentry) &&
            list_empty(&dentry->d_subdirs)) {
-               DPRINTK(("autofs_root_revalidate: dentry=%p %.*s, emptydir\n",
+               DPRINTK(("autofs4_revalidate: dentry=%p %.*s, emptydir\n",
                         dentry, dentry->d_name.len, dentry->d_name.name));
                spin_unlock(&dcache_lock);
-               if (oz_mode)
-                       return 1;
-               else
-                       return try_to_fill_dentry(dentry, dir->i_sb, sbi);
+               if (!oz_mode)
+                       status = try_to_fill_dentry(dentry, dir->i_sb, sbi, flags);
+               return status;
        }
        spin_unlock(&dcache_lock);

@@ -185,25 +402,13 @@
        if (!oz_mode)
                autofs4_update_usage(dentry);

-       return 1;
-}
-
-static int autofs4_revalidate(struct dentry *dentry, int flags)
-{
-       struct autofs_sb_info *sbi = autofs4_sbi(dentry->d_sb);
-
-       if (!autofs4_oz_mode(sbi))
-               autofs4_update_usage(dentry);
-
-       return 1;
+       return status;
 }

 static void autofs4_dentry_release(struct dentry *de)
 {
        struct autofs_info *inf;

-       lock_kernel();
-
        DPRINTK(("autofs4_dentry_release: releasing %p\n", de));

        inf = autofs4_dentry_ino(de);
@@ -215,13 +420,11 @@

                autofs4_free_ino(inf);
        }
-
-       unlock_kernel();
 }

 /* For dentries of directories in the root dir */
 static struct dentry_operations autofs4_root_dentry_operations = {
-       d_revalidate:   autofs4_root_revalidate,
+       d_revalidate:   autofs4_revalidate,
        d_release:      autofs4_dentry_release,
```

```
 };

@@ -236,7 +439,7 @@
 static struct dentry *autofs4_dir_lookup(struct inode *dir, struct dentry *dentry)
 {
 #if 0
-       DPRINTK(("autofs_dir_lookup: ignoring lookup of %.*s/%.*s\n",
+       DPRINTK(("autofs4_dir_lookup: ignoring lookup of %.*s/%.*s\n",
                 dentry->d_parent->d_name.len, dentry->d_parent->d_name.name,
                 dentry->d_name.len, dentry->d_name.name));
 #endif
@@ -252,7 +455,7 @@
        struct autofs_sb_info *sbi;
        int oz_mode;

-       DPRINTK(("autofs_root_lookup: name = %.*s\n",
+       DPRINTK(("autofs4_root_lookup: name = %.*s\n",
                 dentry->d_name.len, dentry->d_name.name));

        if (dentry->d_name.len > NAME_MAX)
@@ -261,7 +464,8 @@
        sbi = autofs4_sbi(dir->i_sb);

        oz_mode = autofs4_oz_mode(sbi);
-       DPRINTK(("autofs_lookup: pid = %u, pgrp = %u, catatonic = %d, oz_mode = %d\n",
+
+       DPRINTK(("autofs4_root_lookup: pid = %u, pgrp = %u, catatonic = %d, oz_mode = %d\n",
                 current->pid, current->pgrp, sbi->catatonic, oz_mode));

        /*
@@ -292,8 +496,15 @@
         * a signal. If so we can force a restart..
         */
        if (dentry->d_flags & DCACHE_AUTOFS_PENDING) {
-               if (signal_pending(current))
-                       return ERR_PTR(-ERESTARTNOINTR);
+               /* See if we were interrupted */
+               if (signal_pending(current)) {
+                       sigset_t *sigset = &current->pending.signal;
+                       if (sigismember (sigset, SIGKILL) ||
+                           sigismember (sigset, SIGQUIT) ||
+                           sigismember (sigset, SIGINT)) {
+                           return ERR_PTR(-ERESTARTNOINTR);
+                       }
+               }
        }

        /*
@@ -317,7 +528,7 @@
        struct inode *inode;
        char *cp;

-       DPRINTK(("autofs_dir_symlink: %s <- %.*s\n", symname,
+       DPRINTK(("autofs4_dir_symlink: %s <- %.*s\n", symname,
                 dentry->d_name.len, dentry->d_name.name));

        if (!autofs4_oz_mode(sbi))
@@ -367,7 +578,7 @@
  * If a process is blocked on the dentry waiting for the expire to finish,
  * it will invalidate the dentry and try to mount with a new one.
```

```
  *
- * Also see autofs_dir_rmdir()..
+ * Also see autofs4_dir_rmdir()..
  */
 static int autofs4_dir_unlink(struct inode *dir, struct dentry *dentry)
 {
@@ -417,8 +628,6 @@
        return 0;
 }

-
-
 static int autofs4_dir_mkdir(struct inode *dir, struct dentry *dentry, int mode)
 {
        struct autofs_sb_info *sbi = autofs4_sbi(dir->i_sb);
@@ -428,7 +637,7 @@
        if ( !autofs4_oz_mode(sbi) )
                return -EACCES;

-       DPRINTK(("autofs_dir_mkdir: dentry %p, creating %.*s\n",
+       DPRINTK(("autofs4_dir_mkdir: dentry %p, creating %.*s\n",
                 dentry, dentry->d_name.len, dentry->d_name.name));

        ino = autofs4_init_ino(ino, sbi, S_IFDIR | 0555);
@@ -452,6 +661,19 @@
        return 0;
 }

+/*
+ * Identify autofs_dentries - this is so we can tell if there's
+ * an extra dentry refcount or not.  We only hold a refcount on the
+ * dentry if its non-negative (ie, d_inode != NULL)
+ */
+int is_autofs4_dentry(struct dentry *dentry)
+{
+       return dentry && dentry->d_inode &&
+               (dentry->d_op == &autofs4_root_dentry_operations ||
+                dentry->d_op == &autofs4_dentry_operations) &&
+               dentry->d_fsdata != NULL;
+}
+
 /* Get/set timeout ioctl() operation */
 static inline int autofs4_get_set_timeout(struct autofs_sb_info *sbi,
                                          unsigned long *p)
@@ -477,16 +699,65 @@
        return put_user(sbi->version, p);
 }

-/* Identify autofs_dentries - this is so we can tell if there's
-   an extra dentry refcount or not.  We only hold a refcount on the
-   dentry if its non-negative (ie, d_inode != NULL)
-*/
-int is_autofs4_dentry(struct dentry *dentry)
+/* Return protocol sub version */
+static inline int autofs4_get_protosubver(struct autofs_sb_info *sbi, int *p)
 {
-       return dentry && dentry->d_inode &&
-               (dentry->d_op == &autofs4_root_dentry_operations ||
-                dentry->d_op == &autofs4_dentry_operations) &&
-               dentry->d_fsdata != NULL;
```

```
+       return put_user(sbi->sub_version, p);
+}
+
+/*
+ * Tells the daemon whether we need to reghost or not. Also, clears
+ * the reghost_needed flag.
+ */
+static inline int autofs4_ask_reghost(struct autofs_sb_info *sbi, int *p)
+{
+       int status;
+
+       DPRINTK(("autofs_ask_reghost: returning %d\n", sbi->needs_reghost));
+
+       status = put_user(sbi->needs_reghost, p);
+       if ( status )
+               return status;
+
+        sbi->needs_reghost = 0;
+        return 0;
+}
+
+/*
+ * Enable / Disable reghosting ioctl() operation
+ */
+static inline int autofs4_toggle_reghost(struct autofs_sb_info *sbi, int *p)
+{
+       int status;
+       int val;
+
+       status = get_user(val, p);
+
+       DPRINTK(("autofs4_toggle_reghost: reghost = %d\n", val));
+
+       if (status)
+               return status;
+
+       /* turn on/off reghosting, with the val */
+       sbi->reghost_enabled = val;
+       return 0;
+}
+
+/*
+ * Tells the daemon whether we can umaont the autofs mount.
+ */
+static inline int autofs4_ask_umount(struct vfsmount *mnt, int *p)
+{
+       int status = 0;
+
+       if (may_umount(mnt) == 0)
+               status = 1;
+
+       DPRINTK(("autofs_ask_umount: returning %d\n", status));
+
+       status = put_user(status, p);
+
+       return status;
 }

 /*
@@ -498,7 +769,7 @@
```

```
 {
        struct autofs_sb_info *sbi = autofs4_sbi(inode->i_sb);

-       DPRINTK(("autofs_ioctl: cmd = 0x%08x, arg = 0x%08lx, sbi = %p, pgrp = %u\n",
+       DPRINTK(("autofs4_ioctl: cmd = 0x%08x, arg = 0x%08lx, sbi = %p, pgrp = %u\n",
                 cmd,arg,sbi,current->pgrp));

        if ( _IOC_TYPE(cmd) != _IOC_TYPE(AUTOFS_IOC_FIRST) ||
@@ -518,9 +789,19 @@
                return 0;
        case AUTOFS_IOC_PROTOVER: /* Get protocol version */
                return autofs4_get_protover(sbi, (int *)arg);
+       case AUTOFS_IOC_PROTOSUBVER: /* Get protocol sub version */
+               return autofs4_get_protosubver(sbi, (int *) arg);
        case AUTOFS_IOC_SETTIMEOUT:
                return autofs4_get_set_timeout(sbi,(unsigned long *)arg);

+       case AUTOFS_IOC_TOGGLEREGHOST:
+               return autofs4_toggle_reghost(sbi, (int *) arg);
+       case AUTOFS_IOC_ASKREGHOST:
+               return autofs4_ask_reghost(sbi, (int *) arg);
+
+       case AUTOFS_IOC_ASKUMOUNT:
+               return autofs4_ask_umount(filp->f_vfsmnt, (int *) arg);
+
        /* return a single thing to expire */
        case AUTOFS_IOC_EXPIRE:
                return autofs4_expire_run(inode->i_sb,filp->f_vfsmnt,sbi,
diff -Nur linux-2.4.18/fs/autofs4/waitq.c autofs4-2.4/fs/autofs4/waitq.c
--- linux-2.4.18/fs/autofs4/waitq.c     2001-02-10 03:29:44.000000000 +0800
+++ autofs4-2.4/fs/autofs4/waitq.c      2004-05-08 12:45:21.000000000 +0800
@@ -3,6 +3,7 @@
  * linux/fs/autofs/waitq.c
  *
  *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
+ *  Copyright 2001-2003 Ian Kent <raven@themaw.net>
  *
  * This file is part of the Linux kernel and is made available under
  * the terms of the GNU General Public License, version 2, or at your
@@ -27,7 +28,7 @@
 {
        struct autofs_wait_queue *wq, *nwq;

-       DPRINTK(("autofs: entering catatonic mode\n"));
+       DPRINTK(("autofs4: entering catatonic mode\n"));

        sbi->catatonic = 1;
        wq = sbi->queues;
@@ -35,9 +36,11 @@
        while ( wq ) {
                nwq = wq->next;
                wq->status = -ENOENT; /* Magic is gone - report failure */
-               kfree(wq->name);
-               wq->name = NULL;
-               wake_up(&wq->queue);
+               if ( wq->name ) {
+                       kfree(wq->name);
+                       wq->name = NULL;
+               }
+               wake_up_interruptible(&wq->queue);
```

```
                wq = nwq;
        }
        if (sbi->pipe) {
@@ -90,8 +93,8 @@
        union autofs_packet_union pkt;
        size_t pktsz;

-       DPRINTK(("autofs_notify: wait id = 0x%08lx, name = %.*s, type=%d\n",
-                wq->wait_queue_token, wq->len, wq->name, type));
+       DPRINTK(("autofs4_notify: wait id = 0x%08lx, name = %.*s, type=%d\n",
+                (unsigned long) wq->wait_queue_token, wq->len, wq->name, type));

        memset(&pkt,0,sizeof pkt); /* For security reasons */

@@ -116,7 +119,7 @@
                memcpy(ep->name, wq->name, wq->len);
                ep->name[wq->len] = '\0';
        } else {
-               printk("autofs_notify_daemon: bad type %d!\n", type);
+               printk("autofs4_notify_daemon: bad type %d!\n", type);
                return;
        }

@@ -124,62 +127,107 @@
                autofs4_catatonic_mode(sbi);
 }

-int autofs4_wait(struct autofs_sb_info *sbi, struct qstr *name,
+static int autofs4_getpath(struct autofs_sb_info *sbi,
+                           struct dentry *dentry, char **name)
+{
+       struct dentry *root = sbi->sb->s_root;
+       struct dentry *tmp;
+       char *buf = *name;
+       char *p;
+       int len = 0;
+
+       spin_lock(&dcache_lock);
+       for (tmp = dentry ; tmp != root ; tmp = tmp->d_parent)
+               len += tmp->d_name.len + 1;
+
+       if (--len > NAME_MAX) {
+               spin_unlock(&dcache_lock);
+               return 0;
+       }
+
+       *(buf + len) = '\0';
+       p = buf + len - dentry->d_name.len;
+       strncpy(p, dentry->d_name.name, dentry->d_name.len);
+
+       for (tmp = dentry->d_parent; tmp != root ; tmp = tmp->d_parent) {
+               *(--p) = '/';
+               p -= tmp->d_name.len;
+               strncpy(p, tmp->d_name.name, tmp->d_name.len);
+       }
+       spin_unlock(&dcache_lock);
+
+       return len;
+}
+
```

```
+int autofs4_wait(struct autofs_sb_info *sbi, struct dentry *dentry,
                enum autofs_notify notify)
 {
        struct autofs_wait_queue *wq;
-       int status;
+       char *name;
+       int len, status;

        /* In catatonic mode, we don't wait for nobody */
-       if ( sbi->catatonic )
+       if (sbi->catatonic)
                return -ENOENT;
-
-       /* We shouldn't be able to get here, but just in case */
-       if ( name->len > NAME_MAX )
+
+       name = kmalloc(NAME_MAX + 1, GFP_KERNEL);
+       if (!name)
+               return -ENOMEM;
+
+       len = autofs4_getpath(sbi, dentry, &name);
+       if (!len) {
+               kfree(name);
                return -ENOENT;
+       }
+
+       if (down_interruptible(&sbi->wq_sem)) {
+               kfree(name);
+               return -EINTR;
+       }

        for ( wq = sbi->queues ; wq ; wq = wq->next ) {
-               if ( wq->hash == name->hash &&
-                    wq->len == name->len &&
-                    wq->name && !memcmp(wq->name,name->name,name->len) )
+               if ( wq->hash == dentry->d_name.hash &&
+                    wq->len == len &&
+                    wq->name && !memcmp(wq->name, name, len) )
                        break;
        }

        if ( !wq ) {
                /* Create a new wait queue */
-               wq = kmalloc(sizeof(struct autofs_wait_queue),GFP_KERNEL);
-               if ( !wq )
-                       return -ENOMEM;
-
-               wq->name = kmalloc(name->len,GFP_KERNEL);
-               if ( !wq->name ) {
-                       kfree(wq);
+               wq = kmalloc(sizeof(struct autofs_wait_queue), GFP_KERNEL);
+               if ( !wq ) {
+                       kfree(name);
+                       up(&sbi->wq_sem);
+                       return -ENOMEM;
                }
+
                wq->wait_queue_token = autofs4_next_wait_queue;
                if (++autofs4_next_wait_queue == 0)
                        autofs4_next_wait_queue = 1;
```

```
-               init_waitqueue_head(&wq->queue);
-               wq->hash = name->hash;
-               wq->len = name->len;
-               wq->status = -EINTR; /* Status return if interrupted */
-               memcpy(wq->name, name->name, name->len);
                wq->next = sbi->queues;
                sbi->queues = wq;
+               init_waitqueue_head(&wq->queue);
+               wq->hash = dentry->d_name.hash;
+               wq->name = name;
+               wq->len = len;
+               wq->status = -EINTR; /* Status return if interrupted */
+               atomic_set(&wq->wait_ctr, 2);
+               up(&sbi->wq_sem);

-               DPRINTK(("autofs_wait: new wait id = 0x%08lx, name = %.*s, nfy=%d\n",
-                        wq->wait_queue_token, wq->len, wq->name, notify));
+               DPRINTK(("autofs4_wait: new wait id = 0x%08lx, name = %.*s, nfy=%d\n",
+                       (unsigned long) wq->wait_queue_token, wq->len, wq->name, notify));
                /* autofs4_notify_daemon() may block */
-               wq->wait_ctr = 2;
                if (notify != NFY_NONE) {
                        autofs4_notify_daemon(sbi,wq,
-                                             notify == NFY_MOUNT ? autofs_ptype_missing :
-                                                                   autofs_ptype_expire_multi)
+                                             notify == NFY_MOUNT ?
+                                                     autofs_ptype_missing :
+                                                     autofs_ptype_expire_multi);
                }
        } else {
-               wq->wait_ctr++;
-               DPRINTK(("autofs_wait: existing wait id = 0x%08lx, name = %.*s, nfy=%d\n",
-                        wq->wait_queue_token, wq->len, wq->name, notify));
+               atomic_inc(&wq->wait_ctr);
+               up(&sbi->wq_sem);
+               DPRINTK(("autofs4_wait: existing wait id = 0x%08lx, name = %.*s, nfy=%d\n",
+                       (unsigned long) wq->wait_queue_token, wq->len, wq->name, notify));
        }

        /* wq->name is NULL if and only if the lock is already released */
@@ -204,7 +252,7 @@
                recalc_sigpending(current);
                spin_unlock_irqrestore(&current->sigmask_lock, irqflags);

-               interruptible_sleep_on(&wq->queue);
+               wait_event_interruptible(wq->queue, wq->name == NULL);

                spin_lock_irqsave(&current->sigmask_lock, irqflags);
                current->blocked = oldset;
@@ -216,7 +264,8 @@

        status = wq->status;

-       if (--wq->wait_ctr == 0)        /* Are we the last process to need status? */
+       /* Are we the last process to need status? */
+       if (atomic_dec_and_test(&wq->wait_ctr))
                kfree(wq);

        return status;
@@ -227,23 +276,29 @@
```

```
 {
        struct autofs_wait_queue *wq, **wql;

+       down(&sbi->wq_sem);
        for ( wql = &sbi->queues ; (wq = *wql) ; wql = &wq->next ) {
                if ( wq->wait_queue_token == wait_queue_token )
                        break;
        }
-       if ( !wq )
+
+       if ( !wq ) {
+               up(&sbi->wq_sem);
                return -EINVAL;
+       }

        *wql = wq->next;        /* Unlink from chain */
+       up(&sbi->wq_sem);
        kfree(wq->name);
        wq->name = NULL;        /* Do not wait on this queue */

        wq->status = status;

-       if (--wq->wait_ctr == 0)        /* Is anyone still waiting for this guy? */
+       /* Is anyone still waiting for this guy? */
+       if (atomic_dec_and_test(&wq->wait_ctr))
                kfree(wq);
        else
-               wake_up(&wq->queue);
+               wake_up_interruptible(&wq->queue);

        return 0;
 }
diff -Nur linux-2.4.18/include/linux/auto_fs.h autofs4-2.4/include/linux/auto_fs.h
--- linux-2.4.18/include/linux/auto_fs.h        2002-04-18 19:32:53.000000000 +0800
+++ autofs4-2.4/include/linux/auto_fs.h 2004-05-08 12:45:20.000000000 +0800
@@ -45,7 +45,7 @@
  * If so, 32-bit user-space code should be backwards compatible.
  */

-#if defined(__sparc__) || defined(__mips__) || defined(__s390__)
+#if defined(__sparc__) || defined(__mips__) || defined(__s390__) || defined(__powerpc__) || d
 typedef unsigned int autofs_wqt_t;
 #else
 typedef unsigned long autofs_wqt_t;
diff -Nur linux-2.4.18/include/linux/auto_fs4.h autofs4-2.4/include/linux/auto_fs4.h
--- linux-2.4.18/include/linux/auto_fs4.h       2002-04-18 19:33:01.000000000 +0800
+++ autofs4-2.4/include/linux/auto_fs4.h        2004-05-08 12:45:20.000000000 +0800
@@ -23,6 +23,12 @@
 #define AUTOFS_MIN_PROTO_VERSION       3
 #define AUTOFS_MAX_PROTO_VERSION       4

+#define AUTOFS_PROTO_SUBVERSION                5
+
+/* Mask for expire behaviour */
+#define AUTOFS_EXP_IMMEDIATE            1
+#define AUTOFS_EXP_LEAVES               2
+
 /* New message type */
 #define autofs_ptype_expire_multi      2       /* Expire entry (umount request) */
```

```
@@ -41,7 +47,10 @@
        struct autofs_packet_expire_multi expire_multi;
 };

-#define AUTOFS_IOC_EXPIRE_MULTI _IOW(0x93,0x66,int)
-
+#define AUTOFS_IOC_EXPIRE_MULTI                _IOW(0x93,0x66,int)
+#define AUTOFS_IOC_PROTOSUBVER         _IOR(0x93,0x67,int)
+#define AUTOFS_IOC_ASKREGHOST          _IOR(0x93,0x68,int)
+#define AUTOFS_IOC_TOGGLEREGHOST       _IOR(0x93,0x69,int)
+#define AUTOFS_IOC_ASKUMOUNT           _IOR(0x93,0x70,int)

 #endif /* _LINUX_AUTO_FS4_H */
```