```
--- /dev/null   Wed Dec 31 16:00:00 1969
+++ linux/include/linux/auto_fs4.h      Sun Oct 22 16:51:49 2000
@@ -0,0 +1,52 @@
+/* -*- c-mode -*-
+ * linux/include/linux/auto_fs4.h
+ *
+ * Copyright 1999-2000 Jeremy Fitzhardinge <jeremy@goop.org>
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ */
+
+#ifndef _LINUX_AUTO_FS4_H
+#define _LINUX_AUTO_FS4_H
+
+/* Include common v3 definitions */
+#include <linux/auto_fs.h>
+
+/* autofs v4 definitions */
+#undef AUTOFS_PROTO_VERSION
+#undef AUTOFS_MIN_PROTO_VERSION
+#undef AUTOFS_MAX_PROTO_VERSION
+
+#define AUTOFS_PROTO_VERSION           4
+#define AUTOFS_MIN_PROTO_VERSION       3
+#define AUTOFS_MAX_PROTO_VERSION       4
+
+/* New message type */
+#define autofs_ptype_expire_multi      2       /* Expire entry (umount request) */
+
+/* v4 multi expire (via pipe) */
+struct autofs_packet_expire_multi {
+       struct autofs_packet_hdr hdr;
+        autofs_wqt_t wait_queue_token;
+       int len;
+       char name[NAME_MAX+1];
+};
+
+union autofs_packet_union {
+       struct autofs_packet_hdr hdr;
+       struct autofs_packet_missing missing;
+       struct autofs_packet_expire expire;
+       struct autofs_packet_expire_multi expire_multi;
+};
+
+#define AUTOFS_IOC_EXPIRE_MULTI _IOW(0x93,0x66,int)
+
+#ifdef __KERNEL__
+
+int init_autofs4_fs(void);
+
+#endif /* __KERNEL__ */
+
+#endif /* _LINUX_AUTO_FS4_H */
--- 2.2/include/linux/auto_fs.h Tue Mar 21 15:25:36 2000
+++ linux/include/linux/auto_fs.h       Sun Oct 22 16:50:41 2000
@@ -14,13 +14,24 @@
 #ifndef _LINUX_AUTO_FS_H
 #define _LINUX_AUTO_FS_H
```

```
+#ifdef __KERNEL__
 #include <linux/version.h>
 #include <linux/fs.h>
 #include <linux/limits.h>
-#include <linux/ioctl.h>
 #include <asm/types.h>

-#define AUTOFS_PROTO_VERSION 3
+int init_autofs_fs(void);
+
+#endif /* __KERNEL__ */
+
+#include <linux/ioctl.h>
+
+/* This file describes autofs v3 */
+#define AUTOFS_PROTO_VERSION   3
+
+/* Range of protocol versions defined */
+#define AUTOFS_MAX_PROTO_VERSION       AUTOFS_PROTO_VERSION
+#define AUTOFS_MIN_PROTO_VERSION       AUTOFS_PROTO_VERSION

 /*
  * Architectures where both 32- and 64-bit binaries can be executed
@@ -43,14 +54,13 @@
 typedef unsigned long autofs_wqt_t;
 #endif

-enum autofs_packet_type {
-       autofs_ptype_missing,   /* Missing entry (mount request) */
-       autofs_ptype_expire,    /* Expire entry (umount request) */
-};
+/* Packet types */
+#define autofs_ptype_missing   0       /* Missing entry (mount request) */
+#define autofs_ptype_expire    1       /* Expire entry (umount request) */

 struct autofs_packet_hdr {
-       int proto_version;            /* Protocol version */
-       enum autofs_packet_type type; /* Type of packet */
+       int proto_version;              /* Protocol version */
+       int type;                       /* Type of packet */
 };

 struct autofs_packet_missing {
@@ -60,6 +70,7 @@
        char name[NAME_MAX+1];
 };

+/* v3 expire (via ioctl) */
 struct autofs_packet_expire {
        struct autofs_packet_hdr hdr;
        int len;
@@ -72,12 +83,5 @@
 #define AUTOFS_IOC_PROTOVER   _IOR(0x93,0x63,int)
 #define AUTOFS_IOC_SETTIMEOUT _IOWR(0x93,0x64,unsigned long)
 #define AUTOFS_IOC_EXPIRE     _IOR(0x93,0x65,struct autofs_packet_expire)
-
-#ifdef __KERNEL__
-
-/* Init function */
```

```
-int init_autofs_fs(void);
-
-#endif /* __KERNEL__ */

 #endif /* _LINUX_AUTO_FS_H */
--- 2.2/fs/filesystems.c        Tue Mar 21 15:25:08 2000
+++ linux/fs/filesystems.c      Sun Oct 22 16:52:10 2000
@@ -139,6 +139,10 @@
        init_autofs_fs();
 #endif

+#ifdef CONFIG_AUTOFS4_FS
+       init_autofs4_fs();
+#endif
+
 #ifdef CONFIG_ADFS_FS
        init_adfs_fs();
 #endif
--- 2.2/fs/Makefile     Tue Mar 21 15:25:05 2000
+++ linux/fs/Makefile   Sun Oct 22 16:44:18 2000
@@ -18,7 +18,7 @@
 MOD_LIST_NAME := FS_MODULES
 ALL_SUB_DIRS = coda minix ext2 fat msdos vfat proc isofs nfs umsdos ntfs \
                hpfs sysv smbfs ncpfs ufs affs romfs autofs hfs lockd \
-               nfsd nls devpts adfs qnx4 efs
+               nfsd nls devpts adfs qnx4 efs autofs4

 ifeq ($(CONFIG_QUOTA),y)
 O_OBJS += dquot.o
@@ -228,6 +228,14 @@
 else
   ifeq ($(CONFIG_AUTOFS_FS),m)
   MOD_SUB_DIRS += autofs
+  endif
+endif
+
+ifeq ($(CONFIG_AUTOFS4_FS),y)
+SUB_DIRS += autofs4
+else
+  ifeq ($(CONFIG_AUTOFS4_FS),m)
+  MOD_SUB_DIRS += autofs4
   endif
 endif

--- 2.2/fs/Config.in    Thu May 11 23:04:48 2000
+++ linux/fs/Config.in  Sun Oct 22 16:44:57 2000
@@ -7,8 +7,8 @@
 bool    'Quota support' CONFIG_QUOTA
 tristate 'Kernel automounter support' CONFIG_AUTOFS_FS

-
 if [ "$CONFIG_EXPERIMENTAL" = "y" ]; then
+  tristate 'Kernel automounter v4 support (supports v3)' CONFIG_AUTOFS4_FS
   tristate 'ADFS filesystem support (read only) (EXPERIMENTAL)' CONFIG_ADFS_FS
 fi
 tristate 'Amiga FFS filesystem support' CONFIG_AFFS_FS
diff -X diffexcl -Nur 2.2/fs/autofs4/Makefile autofs-2.2/fs/autofs4/Makefile
--- 2.2/fs/autofs4/Makefile     Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/Makefile   Mon Oct 23 14:17:58 2000
@@ -0,0 +1,35 @@
```

```
+#
+# Makefile for the linux autofs-filesystem routines.
+#
+# We can build this either out of the kernel tree or the autofs tools tree.
+#
+
+O_TARGET := autofs4.o
+O_OBJS   := init.o inode.o root.o symlink.o waitq.o expire.o
+
+M_OBJS   := $(O_TARGET)
+
+ifdef TOPDIR
+#
+# Part of the kernel code
+#
+include $(TOPDIR)/Rules.make
+else
+#
+# Standalone (handy for development)
+#
+include ../Makefile.rules
+
+CFLAGS += -D__KERNEL__ -DMODULE $(KFLAGS) -I../include -I$(KINCLUDE) $(MODFLAGS)
+
+all: $(O_TARGET)
+
+$(O_TARGET): $(O_OBJS)
+       $(LD) -r -o $(O_TARGET) $(O_OBJS)
+
+install: $(O_TARGET)
+       install -c $(O_TARGET) /lib/modules/‘uname -r‘/fs
+
+clean:
+       rm -f *.o *.s
+endif
diff -X diffexcl -Nur 2.2/fs/autofs4/autofs_i.h autofs-2.2/fs/autofs4/autofs_i.h
--- 2.2/fs/autofs4/autofs_i.h   Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/autofs_i.h Mon Oct 23 14:17:58 2000
@@ -0,0 +1,158 @@
+/* -*- c -*- --------------------------------------------------------------- *
+ *
+ * linux/fs/autofs/autofs_i.h
+ *
+ *   Copyright 1997-1998 Transmeta Corporation - All Rights Reserved
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ *
+ * ------------------------------------------------------------------------- */
+
+/* Internal header file for autofs */
+
+#include <linux/auto_fs4.h>
+#include <linux/list.h>
+
+/* This is the range of ioctl() numbers we claim as ours */
+#define AUTOFS_IOC_FIRST     AUTOFS_IOC_READY
+#define AUTOFS_IOC_COUNT     32
+
```

```
+#include <linux/kernel.h>
+#include <linux/malloc.h>
+#include <linux/sched.h>
+#include <linux/string.h>
+#include <linux/wait.h>
+#include <asm/uaccess.h>
+
+/* #define DEBUG */
+
+#ifdef DEBUG
+#define DPRINTK(D) do{ printk("pid %d: ", current->pid); printk D; } while(0)
+#else
+#define DPRINTK(D) do {} while(0)
+#endif
+
+#define AUTOFS_SUPER_MAGIC 0x0187
+
+/*
+ * If the daemon returns a negative response (AUTOFS_IOC_FAIL) then the
+ * kernel will keep the negative response cached for up to the time given
+ * here, although the time can be shorter if the kernel throws the dcache
+ * entry away.  This probably should be settable from user space.
+ */
+#define AUTOFS_NEGATIVE_TIMEOUT (60*HZ)        /* 1 minute */
+
+/* Unified info structure.  This is pointed to by both the dentry and
+   inode structures.  Each file in the filesystem has an instance of this
+   structure.  It holds a reference to the dentry, so dentries are never
+   flushed while the file exists.  All name lookups are dealt with at the
+   dentry level, although the filesystem can interfere in the validation
+   process.  Readdir is implemented by traversing the dentry lists. */
+struct autofs_info {
+       struct dentry   *dentry;
+       struct inode    *inode;
+
+       int             flags;
+
+       struct autofs_sb_info *sbi;
+       unsigned long last_used;
+
+       mode_t  mode;
+       size_t  size;
+
+       void (*free)(struct autofs_info *);
+       union {
+               const char *symlink;
+       } u;
+};
+
+#define AUTOFS_INF_EXPIRING    (1<<0) /* dentry is in the process of expiring */
+
+struct autofs_wait_queue {
+       struct wait_queue *queue;
+       struct autofs_wait_queue *next;
+       autofs_wqt_t wait_queue_token;
+       /* We use the following to see what we are waiting for */
+       int hash;
+       int len;
+       char *name;
+       /* This is for status reporting upon return */
```

```
+	int status;
+	int wait_ctr;
+};
+
+#define AUTOFS_SBI_MAGIC 0x6d4a556d
+
+struct autofs_sb_info {
+	u32 magic;
+	struct file *pipe;
+	pid_t oz_pgrp;
+	int catatonic;
+	int version;
+	unsigned long exp_timeout;
+	struct super_block *sb;
+	struct autofs_wait_queue *queues; /* Wait queue pointer */
+};
+
+static inline struct autofs_sb_info *autofs4_sbi(struct super_block *sb)
+{
+	return (struct autofs_sb_info *)(sb->u.generic_sbp);
+}
+
+static inline struct autofs_info *autofs4_dentry_ino(struct dentry *dentry)
+{
+	return (struct autofs_info *)(dentry->d_fsdata);
+}
+
+/* autofs4_oz_mode(): do we see the man behind the curtain?  (The
+   processes which do manipulations for us in user space sees the raw
+   filesystem without "magic".) */
+
+static inline int autofs4_oz_mode(struct autofs_sb_info *sbi) {
+	return sbi->catatonic || current->pgrp == sbi->oz_pgrp;
+}
+
+/* Does a dentry have some pending activity? */
+static inline int autofs4_ispending(struct dentry *dentry)
+{
+	struct autofs_info *inf = autofs4_dentry_ino(dentry);
+
+	return (dentry->d_flags & DCACHE_AUTOFS_PENDING) ||
+		(inf != NULL && inf->flags & AUTOFS_INF_EXPIRING);
+}
+
+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 *);
+
+/* Expiration */
+int is_autofs4_dentry(struct dentry *);
+int autofs4_expire_run(struct super_block *, struct autofs_sb_info *,
+		       struct autofs_packet_expire *);
+int autofs4_expire_multi(struct super_block *, struct autofs_sb_info *, int *);
+
+/* Operations structures */
+
+extern struct inode_operations autofs4_symlink_inode_operations;
+extern struct inode_operations autofs4_dir_inode_operations;
+extern struct inode_operations autofs4_root_inode_operations;
+
```

```
+/* Initializing function */
+
+struct super_block *autofs4_read_super(struct super_block *, void *,int);
+struct autofs_info *autofs4_init_ino(struct autofs_info *, struct autofs_sb_info *sbi, mode_t
+
+/* Queue management functions */
+
+enum autofs_notify
+{
+       NFY_NONE,
+       NFY_MOUNT,
+       NFY_EXPIRE
+};
+
+int autofs4_wait(struct autofs_sb_info *,struct qstr *, enum autofs_notify);
+int autofs4_wait_release(struct autofs_sb_info *,autofs_wqt_t,int);
+void autofs4_catatonic_mode(struct autofs_sb_info *);
diff -X diffexcl -Nur 2.2/fs/autofs4/expire.c autofs-2.2/fs/autofs4/expire.c
--- 2.2/fs/autofs4/expire.c     Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/expire.c   Mon Oct 23 14:17:58 2000
@@ -0,0 +1,234 @@
+/* -*- c -*- --------------------------------------------------------------- *
+ *
+ * linux/fs/autofs/expire.c
+ *
+ *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
+ *  Copyright 1999-2000 Jeremy Fitzhardinge <jeremy@goop.org>
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ *
+ * ------------------------------------------------------------------------- */
+
+#include "autofs_i.h"
+
+/*
+ * Determine if a dentry tree is in use.  This is much the
+ * same as the standard is_root_busy() function, except
+ * that :-
+ *  - the extra dentry reference in autofs dentries is not
+ *    considered to be busy
+ *  - mountpoints within the tree are not busy
+ *  - it traverses across mountpoints
+ * XXX doesn’t consider children of covered dentries at mountpoints
+ */
+static int is_tree_busy(struct dentry *root)
+{
+       struct dentry *this_parent;
+       struct list_head *next;
+       int count;
+
+       root = root->d_mounts;
+
+       count = root->d_count;
+       this_parent = root;
+
+       DPRINTK(("is_tree_busy: starting at %.*s/%.*s, d_count=%d\n",
+                root->d_covers->d_parent->d_name.len,
+                root->d_covers->d_parent->d_name.name,
```

```
+                root->d_name.len, root->d_name.name,
+                root->d_count));
+
+       /* Ignore autofs's extra reference */
+       if (is_autofs4_dentry(root)) {
+               DPRINTK(("is_tree_busy: autofs\n"));
+               count--;
+       }
+
+       /* Mountpoints don't count */
+       if (root->d_mounts != root ||
+           root->d_covers != root) {
+               DPRINTK(("is_tree_busy: mountpoint\n"));
+               count--;
+       }
+
+repeat:
+       next = this_parent->d_mounts->d_subdirs.next;
+resume:
+       while (next != &this_parent->d_mounts->d_subdirs) {
+               int adj = 0;
+               struct list_head *tmp = next;
+               struct dentry *dentry = list_entry(tmp, struct dentry,
+                                                  d_child);
+
+               next = tmp->next;
+
+               dentry = dentry->d_mounts;
+
+               DPRINTK(("is_tree_busy: considering %.*s/%.*s, d_count=%d, count=%d\n",
+                        this_parent->d_name.len,
+                        this_parent->d_name.name,
+                        dentry->d_covers->d_name.len,
+                        dentry->d_covers->d_name.name,
+                        dentry->d_count, count));
+
+               /* Decrement count for unused children */
+               count += (dentry->d_count - 1);
+
+               /* Mountpoints don't count */
+               if (dentry->d_mounts != dentry ||
+                   dentry->d_covers != dentry) {
+                       DPRINTK(("is_tree_busy: mountpoint\n"));
+                       adj++;
+               }
+
+               /* Ignore autofs's extra reference */
+               if (is_autofs4_dentry(dentry)) {
+                       DPRINTK(("is_tree_busy: autofs\n"));
+                       adj++;
+               }
+
+               count -= adj;
+
+               if (!list_empty(&dentry->d_mounts->d_subdirs)) {
+                       this_parent = dentry->d_mounts;
+                       goto repeat;
+               }
+
+               /* root is busy if any leaf is busy */
```

```
+               if (dentry->d_count != adj) {
+                       DPRINTK(("is_tree_busy: busy leaf (d_count=%d adj=%d)\n",
+                                dentry->d_count, adj));
+                       return 1;
+               }
+       }
+       /*
+        * All done at this level ... ascend and resume the search.
+        */
+       if (this_parent != root) {
+               next = this_parent->d_covers->d_child.next;
+               this_parent = this_parent->d_covers->d_parent;
+               goto resume;
+       }
+
+       DPRINTK(("is_tree_busy: count=%d\n", count));
+       return count != 0; /* remaining users? */
+}
+
+/*
+ * Find an eligible tree to time-out
+ * A tree is eligible if :-
+ *  - it is unused by any user process
+ *  - it has been unused for exp_timeout time
+ */
+static struct dentry *autofs4_expire(struct super_block *sb,
+                                    struct autofs_sb_info *sbi,
+                                    int do_now)
+{
+       unsigned long now = jiffies; /* snapshot of now */
+       unsigned long timeout;
+       struct dentry *root = sb->s_root;
+       struct list_head *tmp;
+
+       if (!sbi->exp_timeout || !root)
+               return NULL;
+
+       timeout = sbi->exp_timeout;
+
+       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);
+
+               if (dentry->d_inode == NULL)
+                       continue;
+
+               ino = autofs4_dentry_ino(dentry);
+
+               if (ino == NULL) {
+                       /* dentry in the process of being deleted */
+                       continue;
+               }
+
+               /* No point expiring a pending mount */
+               if (dentry->d_flags & DCACHE_AUTOFS_PENDING)
+                       continue;
+
+               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;
+               }
+
+               if (!is_tree_busy(dentry)) {
+                       DPRINTK(("autofs_expire: returning %p %.*s\n",
+                                dentry, 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);
+                       return dentry;
+               }
+       }
+
+       return NULL;
+}
+
+/* Perform an expiry operation */
+int autofs4_expire_run(struct super_block *sb,
+                      struct autofs_sb_info *sbi,
+                      struct autofs_packet_expire *pkt_p)
+{
+       struct autofs_packet_expire pkt;
+       struct dentry *dentry;
+
+       memset(&pkt,0,sizeof pkt);
+
+       pkt.hdr.proto_version = sbi->version;
+       pkt.hdr.type = autofs_ptype_expire;
+
+       if ((dentry = autofs4_expire(sb, sbi, 0)) == NULL)
+               return -EAGAIN;
+
+       pkt.len = dentry->d_name.len;
+       memcpy(pkt.name, dentry->d_name.name, pkt.len);
+       pkt.name[pkt.len] = '\0';
+
+       if ( copy_to_user(pkt_p, &pkt, sizeof(struct autofs_packet_expire)) )
+               return -EFAULT;
+
+       return 0;
+}
+
+/* Call repeatedly until it returns -EAGAIN, meaning there's nothing
+   more to be done */
+int autofs4_expire_multi(struct super_block *sb,
+                       struct autofs_sb_info *sbi, int *arg)
+{
+       struct dentry *dentry;
+       int ret = -EAGAIN;
+       int do_now = 0;
+
+       if (arg && get_user(do_now, arg))
```

```
+               return -EFAULT;
+
+       if ((dentry = autofs4_expire(sb, sbi, do_now)) != NULL) {
+               struct autofs_info *de_info = autofs4_dentry_ino(dentry);
+
+               /* 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);
+               de_info->flags &= ~AUTOFS_INF_EXPIRING;
+       }
+
+       return ret;
+}
+
diff -X diffexcl -Nur 2.2/fs/autofs4/init.c autofs-2.2/fs/autofs4/init.c
--- 2.2/fs/autofs4/init.c       Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/init.c     Sun Oct 22 19:39:16 2000
@@ -0,0 +1,40 @@
+/* -*- c -*- --------------------------------------------------------------- *
+ *
+ * linux/fs/autofs4/init.c
+ *
+ *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ *
+ * ------------------------------------------------------------------------- */
+
+#include <linux/module.h>
+#include <linux/init.h>
+#include "autofs_i.h"
+
+static struct file_system_type autofs4_fs_type = {
+       "autofs",
+       0,
+       autofs4_read_super,
+       NULL
+};
+
+#ifdef MODULE
+int init_module(void)
+{
+       return register_filesystem(&autofs4_fs_type);
+}
+
+int cleanup_module(void)
+{
+       return unregister_filesystem(&autofs4_fs_type);
+}
+#else
+__initfunc(int init_autofs4_fs(void))
+{
+       return register_filesystem(&autofs4_fs_type);
+}
+#endif
+
diff -X diffexcl -Nur 2.2/fs/autofs4/inode.c autofs-2.2/fs/autofs4/inode.c
```

```
--- 2.2/fs/autofs4/inode.c      Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/inode.c    Mon Oct 23 14:17:58 2000
@@ -0,0 +1,382 @@
+/* -*- c -*- --------------------------------------------------------------- *
+ *
+ * linux/fs/autofs/inode.c
+ *
+ *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ *
+ * ------------------------------------------------------------------------- */
+
+#include <linux/kernel.h>
+#include <linux/malloc.h>
+#include <linux/file.h>
+#include <linux/locks.h>
+#include <asm/bitops.h>
+#include "autofs_i.h"
+#define __NO_VERSION__
+#include <linux/module.h>
+
+static void ino_lnkfree(struct autofs_info *ino)
+{
+       if (ino->u.symlink) {
+               kfree(ino->u.symlink);
+               ino->u.symlink = NULL;
+       }
+}
+
+struct autofs_info *autofs4_init_ino(struct autofs_info *ino,
+                                    struct autofs_sb_info *sbi, mode_t mode)
+{
+       int reinit = 1;
+
+       if (ino == NULL) {
+               reinit = 0;
+               ino = kmalloc(sizeof(*ino), GFP_KERNEL);
+       }
+
+       if (ino == NULL)
+               return NULL;
+
+       ino->flags = 0;
+       ino->mode = mode;
+       ino->inode = NULL;
+       ino->dentry = NULL;
+       ino->size = 0;
+
+       ino->last_used = jiffies;
+
+       ino->sbi = sbi;
+
+       if (reinit && ino->free)
+               (ino->free)(ino);
+
+       memset(&ino->u, 0, sizeof(ino->u));
+
```

```
+	ino->free = NULL;
+
+	if (S_ISLNK(mode))
+		ino->free = ino_lnkfree;
+
+	return ino;
+}
+
+void autofs4_free_ino(struct autofs_info *ino)
+{
+	if (ino->dentry) {
+		ino->dentry->d_fsdata = NULL;
+		if (ino->dentry->d_inode)
+			dput(ino->dentry);
+		ino->dentry = NULL;
+	}
+	if (ino->free)
+		(ino->free)(ino);
+	kfree(ino);
+}
+
+static void autofs4_put_super(struct super_block *sb)
+{
+	struct autofs_sb_info *sbi = autofs4_sbi(sb);
+
+	sb->u.generic_sbp = NULL;
+
+	if ( !sbi->catatonic )
+		autofs4_catatonic_mode(sbi); /* Free wait queues, close pipe */
+
+	kfree(sbi);
+
+	DPRINTK(("autofs: shutting down\n"));
+	MOD_DEC_USE_COUNT;
+}
+
+static void autofs4_umount_begin(struct super_block *sb)
+{
+	struct autofs_sb_info *sbi = autofs4_sbi(sb);
+
+	if (!sbi->catatonic)
+		autofs4_catatonic_mode(sbi);
+}
+
+static int autofs4_statfs(struct super_block *sb, struct statfs *buf, int bufsz);
+
+static struct super_operations autofs4_sops = {
+	put_super:	autofs4_put_super,
+	statfs:		autofs4_statfs,
+	umount_begin:	autofs4_umount_begin,
+};
+
+static int parse_options(char *options, int *pipefd, uid_t *uid, gid_t *gid,
+			 pid_t *pgrp, int *minproto, int *maxproto)
+{
+	char *this_char, *value;
+
+	*uid = current->uid;
+	*gid = current->gid;
+	*pgrp = current->pgrp;
```

```
+
+       *minproto = AUTOFS_MIN_PROTO_VERSION;
+       *maxproto = AUTOFS_MAX_PROTO_VERSION;
+
+       *pipefd = -1;
+
+       if ( !options ) return 1;
+       for (this_char = strtok(options,","); this_char; this_char = strtok(NULL,",")) {
+               if ((value = strchr(this_char,'=')) != NULL)
+                       *value++ = 0;
+               if (!strcmp(this_char,"fd")) {
+                       if (!value || !*value)
+                               return 1;
+                       *pipefd = simple_strtoul(value,&value,0);
+                       if (*value)
+                               return 1;
+               }
+               else if (!strcmp(this_char,"uid")) {
+                       if (!value || !*value)
+                               return 1;
+                       *uid = simple_strtoul(value,&value,0);
+                       if (*value)
+                               return 1;
+               }
+               else if (!strcmp(this_char,"gid")) {
+                       if (!value || !*value)
+                               return 1;
+                       *gid = simple_strtoul(value,&value,0);
+                       if (*value)
+                               return 1;
+               }
+               else if (!strcmp(this_char,"pgrp")) {
+                       if (!value || !*value)
+                               return 1;
+                       *pgrp = simple_strtoul(value,&value,0);
+                       if (*value)
+                               return 1;
+               }
+               else if (!strcmp(this_char,"minproto")) {
+                       if (!value || !*value)
+                               return 1;
+                       *minproto = simple_strtoul(value,&value,0);
+                       if (*value)
+                               return 1;
+               }
+               else if (!strcmp(this_char,"maxproto")) {
+                       if (!value || !*value)
+                               return 1;
+                       *maxproto = simple_strtoul(value,&value,0);
+                       if (*value)
+                               return 1;
+               }
+               else break;
+       }
+       return (*pipefd < 0);
+}
+
+static struct autofs_info *autofs4_mkroot(struct autofs_sb_info *sbi)
+{
+       struct autofs_info *ino;
```

```
+
+       ino = autofs4_init_ino(NULL, sbi, S_IFDIR | 0755);
+       if (!ino)
+               return NULL;
+
+       return ino;
+}
+
+struct super_block *autofs4_read_super(struct super_block *s, void *data,
+                                      int silent)
+{
+       struct inode * root_inode;
+       struct dentry * root;
+       struct file * pipe;
+       int pipefd;
+       struct autofs_sb_info *sbi;
+       int minproto, maxproto;
+
+       MOD_INC_USE_COUNT;
+
+       lock_super(s);
+       /* Super block already completed? */
+       if (s->s_root)
+               goto out_unlock;
+
+       sbi = (struct autofs_sb_info *) kmalloc(sizeof(*sbi), GFP_KERNEL);
+       if ( !sbi )
+               goto fail_unlock;
+       DPRINTK(("autofs: starting up, sbi = %p\n",sbi));
+
+       memset(sbi, 0, sizeof(*sbi));
+
+       s->u.generic_sbp = sbi;
+       sbi->magic = AUTOFS_SBI_MAGIC;
+       sbi->catatonic = 0;
+       sbi->exp_timeout = 0;
+       sbi->oz_pgrp = current->pgrp;
+       sbi->sb = s;
+       sbi->version = 0;
+       sbi->queues = NULL;
+       s->s_blocksize = 1024;
+       s->s_blocksize_bits = 10;
+       s->s_magic = AUTOFS_SUPER_MAGIC;
+       s->s_op = &autofs4_sops;
+       s->s_root = NULL;
+       unlock_super(s);
+
+       /*
+        * Get the root inode and dentry, but defer checking for errors.
+        */
+       root_inode = autofs4_get_inode(s, autofs4_mkroot(sbi));
+       root_inode->i_op = &autofs4_root_inode_operations;
+       root = d_alloc_root(root_inode, NULL);
+       pipe = NULL;
+
+       /*
+        * Check whether somebody else completed the super block.
+        */
+       if (s->s_root)
+               goto out_dput;
```

```
+
+       if (!root)
+               goto fail_iput;
+
+       /* Can this call block? */
+       if (parse_options(data, &pipefd,
+                         &root_inode->i_uid, &root_inode->i_gid,
+                         &sbi->oz_pgrp,
+                         &minproto, &maxproto)) {
+               printk("autofs: 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 "
+                      "daemon (%d, %d) kernel (%d, %d)\n",
+                       minproto, maxproto,
+                       AUTOFS_MIN_PROTO_VERSION, AUTOFS_MAX_PROTO_VERSION);
+               goto fail_dput;
+       }
+
+       sbi->version = maxproto > AUTOFS_MAX_PROTO_VERSION ? AUTOFS_MAX_PROTO_VERSION : maxpro
+
+       DPRINTK(("autofs: pipe fd = %d, pgrp = %u\n", pipefd, sbi->oz_pgrp));
+       pipe = fget(pipefd);
+       /*
+        * Check whether somebody else completed the super block.
+        */
+       if (s->s_root)
+               goto out_fput;
+
+       if ( !pipe ) {
+               printk("autofs: could not open pipe file descriptor\n");
+               goto fail_dput;
+       }
+       if ( !pipe->f_op || !pipe->f_op->write )
+               goto fail_fput;
+       sbi->pipe = pipe;
+
+       /*
+        * Success! Install the root dentry now to indicate completion.
+        */
+       s->s_root = root;
+       return s;
+
+       /*
+        * Success ... somebody else completed the super block for us.
+        */
+out_unlock:
+       unlock_super(s);
+       goto out_dec;
+out_fput:
+       if (pipe)
+               fput(pipe);
+out_dput:
+       if (root)
+               dput(root);
+       else
```

```
+               iput(root_inode);
+out_dec:
+       MOD_DEC_USE_COUNT;
+       return s;
+
+       /*
+        * Failure ... clear the s_dev slot and clean up.
+        */
+fail_fput:
+       printk("autofs: pipe file descriptor does not contain proper ops\n");
+       /*
+        * fput() can block, so we clear the super block first.
+        */
+       fput(pipe);
+       /* fall through */
+fail_dput:
+       /*
+        * dput() can block, so we clear the super block first.
+        */
+       dput(root);
+       goto fail_free;
+fail_iput:
+       printk("autofs: get root dentry failed\n");
+       /*
+        * iput() can block, so we clear the super block first.
+        */
+       iput(root_inode);
+fail_free:
+       kfree(sbi);
+fail_unlock:
+       s->s_dev = 0;
+       MOD_DEC_USE_COUNT;
+       return NULL;
+}
+
+static int autofs4_statfs(struct super_block *sb, struct statfs *buf, int bufsiz)
+{
+       struct statfs tmp;
+
+       tmp.f_type = AUTOFS_SUPER_MAGIC;
+       tmp.f_bsize = 1024;
+       tmp.f_blocks = 0;
+       tmp.f_bfree = 0;
+       tmp.f_bavail = 0;
+       tmp.f_files = 0;
+       tmp.f_ffree = 0;
+       tmp.f_namelen = NAME_MAX;
+       return copy_to_user(buf, &tmp, bufsiz) ? -EFAULT : 0;
+}
+
+struct inode *autofs4_get_inode(struct super_block *sb,
+                                struct autofs_info *inf)
+{
+       struct inode *inode = get_empty_inode();
+
+       if (inode == NULL)
+               return NULL;
+
+       inf->inode = inode;
+       inode->i_sb = sb;
```

```
+       inode->i_dev = sb->s_dev;
+       inode->i_mode = inf->mode;
+       if (sb->s_root) {
+               inode->i_uid = sb->s_root->d_inode->i_uid;
+               inode->i_gid = sb->s_root->d_inode->i_gid;
+       } else {
+               inode->i_uid = 0;
+               inode->i_gid = 0;
+       }
+       inode->i_size = 0;
+       inode->i_blksize = PAGE_CACHE_SIZE;
+       inode->i_blocks = 0;
+       inode->i_rdev = 0;
+       inode->i_nlink = 1;
+       inode->i_op = NULL;
+       inode->i_atime = inode->i_mtime = inode->i_ctime = CURRENT_TIME;
+
+       if (S_ISDIR(inf->mode)) {
+               inode->i_nlink = 2;
+               inode->i_op = &autofs4_dir_inode_operations;
+       } else if (S_ISLNK(inf->mode))
+               inode->i_op = &autofs4_symlink_inode_operations;
+
+       return inode;
+}
diff -X diffexcl -Nur 2.2/fs/autofs4/inohash.c autofs-2.2/fs/autofs4/inohash.c
--- 2.2/fs/autofs4/inohash.c    Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/inohash.c  Sun Oct 22 16:46:18 2000
@@ -0,0 +1,68 @@
+/*
+ * "inohash" is a misnomer.  Inodes are just stored in a single list,
+ * since this code is only ever asked for the most recently inserted
+ * inode.
+ *
+ *   Copyright 1999 Jeremy Fitzhardinge <jeremy@goop.org>
+ */
+
+#include "autofs_i.h"
+
+void autofs4_init_ihash(struct autofs_inohash *ih)
+{
+       INIT_LIST_HEAD(&ih->head);
+}
+
+void autofs4_ihash_insert(struct autofs_inohash *ih,
+                         struct autofs_info *ino)
+{
+       DPRINTK(("autofs_ihash_insert: adding ino %ld\n", ino->ino));
+
+       list_add(&ino->ino_hash, &ih->head);
+}
+
+void autofs4_ihash_delete(struct autofs_info *inf)
+{
+       DPRINTK(("autofs_ihash_delete: deleting ino %ld\n", inf->ino));
+
+       if (!list_empty(&inf->ino_hash))
+               list_del(&inf->ino_hash);
+}
+
```

```
+struct autofs_info *autofs4_ihash_find(struct autofs_inohash *ih,
+                                       ino_t inum)
+{
+       struct list_head *tmp;
+
+       for(tmp = ih->head.next;
+           tmp != &ih->head;
+           tmp = tmp->next) {
+               struct autofs_info *ino = list_entry(tmp, struct autofs_info, ino_hash);
+               if (ino->ino == inum) {
+                       DPRINTK(("autofs_ihash_find: found %ld -> %p\n",
+                                inum, ino));
+                       return ino;
+               }
+       }
+       DPRINTK(("autofs_ihash_find: didn't find %ld\n", inum));
+       return NULL;
+}
+
+void autofs4_ihash_nuke(struct autofs_inohash *ih)
+{
+       struct list_head *tmp = ih->head.next;
+       struct list_head *next;
+
+       for(; tmp != &ih->head; tmp = next) {
+               struct autofs_info *ino;
+
+               next = tmp->next;
+
+               ino = list_entry(tmp, struct autofs_info, ino_hash);
+
+               DPRINTK(("autofs_ihash_nuke: nuking %ld\n", ino->ino));
+               autofs4_free_ino(ino);
+       }
+       INIT_LIST_HEAD(&ih->head);
+}
+
diff -X diffexcl -Nur 2.2/fs/autofs4/root.c autofs-2.2/fs/autofs4/root.c
--- 2.2/fs/autofs4/root.c       Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/root.c     Mon Oct 23 14:17:58 2000
@@ -0,0 +1,625 @@
+/* -*- c -*- --------------------------------------------------------------- *
+ *
+ * linux/fs/autofs/root.c
+ *
+ *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
+ *  Copyright 1999-2000 Jeremy Fitzhardinge <jeremy@goop.org>
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ *
+ * ------------------------------------------------------------------------- */
+
+#include <linux/errno.h>
+#include <linux/stat.h>
+#include <linux/param.h>
+#include "autofs_i.h"
+
+static int autofs4_dir_readdir(struct file *,void *,filldir_t);
```

```
+static struct dentry *autofs4_dir_lookup(struct inode *,struct dentry *);
+static int autofs4_dir_symlink(struct inode *,struct dentry *,const char *);
+static int autofs4_dir_unlink(struct inode *,struct dentry *);
+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 struct dentry *autofs4_root_lookup(struct inode *,struct dentry *);
+
+static ssize_t generic_read_dir(struct file *filp, char *buf, size_t siz, loff_t *ppos)
+{
+       return -EISDIR;
+}
+
+static struct file_operations autofs4_root_operations = {
+       read:           generic_read_dir,
+       readdir:        autofs4_dir_readdir,
+       ioctl:          autofs4_root_ioctl,
+};
+
+static struct file_operations autofs4_dir_operations = {
+       read:           generic_read_dir,
+       readdir:        autofs4_dir_readdir,
+};
+
+struct inode_operations autofs4_root_inode_operations = {
+       default_file_ops:&autofs4_root_operations,
+       lookup:         autofs4_root_lookup,
+       unlink:         autofs4_dir_unlink,
+       symlink:        autofs4_dir_symlink,
+       mkdir:          autofs4_dir_mkdir,
+       rmdir:          autofs4_dir_rmdir,
+};
+
+struct inode_operations autofs4_dir_inode_operations = {
+       default_file_ops:&autofs4_dir_operations,
+       lookup:         autofs4_dir_lookup,
+       unlink:         autofs4_dir_unlink,
+       symlink:        autofs4_dir_symlink,
+       mkdir:          autofs4_dir_mkdir,
+       rmdir:          autofs4_dir_rmdir,
+};
+
+static inline struct dentry *nth_child(struct dentry *dir, int nr)
+{
+       struct list_head *tmp = dir->d_subdirs.next;
+
+       while(tmp != &dir->d_subdirs) {
+               if (nr-- == 0)
+                       return list_entry(tmp, struct dentry, d_child);
+               tmp = tmp->next;
+       }
+       return NULL;
+}
+
+static int autofs4_dir_readdir(struct file *filp, void *dirent,
+                              filldir_t filldir)
+{
+       struct autofs_sb_info *sbi;
+       struct autofs_info *ino;
+       struct dentry *dentry = filp->f_dentry;
```

```
+	struct dentry *dent_ptr;
+	struct inode *dir = dentry->d_inode;
+	struct list_head *cursor;
+	off_t nr;
+
+	sbi = autofs4_sbi(dir->i_sb);
+	ino = autofs4_dentry_ino(dentry);
+	nr = filp->f_pos;
+
+	switch(nr)
+	{
+	case 0:
+		if (filldir(dirent, ".", 1, nr, dir->i_ino) < 0)
+			return 0;
+		filp->f_pos = ++nr;
+		/* fall through */
+	case 1:
+		if (filldir(dirent, "..", 2, nr, dentry->d_covers->d_parent->d_inode->i_ino) <
+			return 0;
+		filp->f_pos = ++nr;
+		/* fall through */
+	default:
+		dent_ptr = nth_child(dentry, nr-2);
+		if (dent_ptr == NULL)
+			break;
+
+		cursor = &dent_ptr->d_child;
+
+		while(cursor != &dentry->d_subdirs) {
+			dent_ptr = list_entry(cursor, struct dentry, d_child);
+			if (dent_ptr->d_inode &&
+			    filldir(dirent, dent_ptr->d_name.name, dent_ptr->d_name.len, nr,
+					dent_ptr->d_inode->i_ino) < 0)
+				return 0;
+			filp->f_pos = ++nr;
+			cursor = cursor->next;
+		}
+		break;
+	}
+
+	return 0;
+}
+
+/* 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)
+{
+	struct dentry *top = dentry->d_sb->s_root;
+
+	for(; dentry != top; dentry = dentry->d_parent) {
+		struct autofs_info *ino = autofs4_dentry_ino(dentry->d_covers);
+
+		if (ino) {
+			update_atime(dentry->d_inode);
+			ino->last_used = jiffies;
+		}
+	}
+}
+
+static int try_to_fill_dentry(struct dentry *dentry,
```

```
+                           struct super_block *sb,
+                           struct autofs_sb_info *sbi)
+{
+       struct autofs_info *de_info = autofs4_dentry_ino(dentry);
+       int status = 0;
+
+       /* Block on any pending expiry here; invalidate the dentry
+           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: expire done status=%d\n", status));
+
+               return 0;
+       }
+
+       DPRINTK(("try_to_fill_entry: dentry=%p %.*s ino=%p\n",
+                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);
+
+               DPRINTK(("try_to_fill_entry: mount done status=%d\n", status));
+
+               if (status && dentry->d_inode)
+                       return 0; /* Try to get the kernel to invalidate this dentry */
+
+               /* Turn this into a real negative dentry? */
+               if (status == -ENOENT) {
+                       dentry->d_time = jiffies + AUTOFS_NEGATIVE_TIMEOUT;
+                       dentry->d_flags &= ~DCACHE_AUTOFS_PENDING;
+                       return 1;
+               } else if (status) {
+                       /* Return a negative dentry, but leave it "pending" */
+                       return 1;
+               }
+       }
+
+       /* If this is an unused directory that isn't a mount point,
+          bitch at the daemon and fix it in user space */
+       if (S_ISDIR(dentry->d_inode->i_mode) &&
+           dentry->d_mounts == dentry &&
+           list_empty(&dentry->d_subdirs)) {
+               DPRINTK(("try_to_fill_entry: mounting existing dir\n"));
+               return autofs4_wait(sbi, &dentry->d_name, NFY_MOUNT) == 0;
+       }
+
+       /* We don't update the usages for the autofs daemon itself, this
+          is necessary for recursive autofs mounts */
+       if (!autofs4_oz_mode(sbi))
+               autofs4_update_usage(dentry);
```

```
+
+       dentry->d_flags &= ~DCACHE_AUTOFS_PENDING;
+       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)
+{
+       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);
+
+       /* Pending dentry */
+       if (autofs4_ispending(dentry)) {
+               if (oz_mode)
+                       return 1;
+               else
+                       return try_to_fill_dentry(dentry, dir->i_sb, sbi);
+       }
+
+       /* Negative dentry.. invalidate if "old" */
+       if (dentry->d_inode == NULL)
+               return (dentry->d_time - jiffies <= AUTOFS_NEGATIVE_TIMEOUT);
+
+       ino = autofs4_dentry_ino(dentry);
+
+       /* Check for a non-mountpoint directory with no contents */
+       if (S_ISDIR(dentry->d_inode->i_mode) &&
+           dentry->d_mounts == dentry &&
+           list_empty(&dentry->d_subdirs)) {
+               DPRINTK(("autofs_root_revalidate: dentry=%p %.*s, emptydir\n",
+                        dentry, dentry->d_name.len, dentry->d_name.name));
+               if (oz_mode)
+                       return 1;
+               else
+                       return try_to_fill_dentry(dentry, dir->i_sb, sbi);
+       }
+
+       /* Update the usage list */
+       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;
```

```
+}
+
+static void autofs4_dentry_release(struct dentry *de)
+{
+       struct autofs_info *inf = autofs4_dentry_ino(de);
+
+       DPRINTK(("autofs4_dentry_release: releasing %p\n", de));
+
+       de->d_fsdata = NULL;
+       if (inf) {
+               inf->dentry = NULL;
+               inf->inode = NULL;
+
+               autofs4_free_ino(inf);
+       }
+}
+
+/* For dentries of directories in the root dir */
+static struct dentry_operations autofs4_root_dentry_operations = {
+       d_revalidate:   autofs4_root_revalidate,        /* d_revalidate */
+       d_release:      autofs4_dentry_release,
+};
+
+/* For other dentries */
+static struct dentry_operations autofs4_dentry_operations = {
+       d_revalidate:   autofs4_revalidate,     /* d_revalidate */
+       d_release:      autofs4_dentry_release,
+};
+
+/* Lookups in non-root dirs never find anything - if it’s there, it’s
+   already in the dcache */
+static struct dentry *autofs4_dir_lookup(struct inode *dir, struct dentry *dentry)
+{
+#if 0
+       DPRINTK(("autofs_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
+
+       dentry->d_fsdata = NULL;
+       d_add(dentry, NULL);
+       return NULL;
+}
+
+/* Lookups in the root directory */
+static struct dentry *autofs4_root_lookup(struct inode *dir, struct dentry *dentry)
+{
+       struct autofs_sb_info *sbi;
+       int oz_mode;
+
+       DPRINTK(("autofs_root_lookup: name = %.*s\n",
+                dentry->d_name.len, dentry->d_name.name));
+
+       if (dentry->d_name.len > NAME_MAX)
+               return ERR_PTR(-ENAMETOOLONG);/* File name too long to exist */
+
+       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",
```

```
+                current->pid, current->pgrp, sbi->catatonic, oz_mode));
+
+       /*
+        * Mark the dentry incomplete, but add it. This is needed so
+        * that the VFS layer knows about the dentry, and we can count
+        * on catching any lookups through the revalidate.
+        *
+        * Let all the hard work be done by the revalidate function that
+        * needs to be able to do this anyway..
+        *
+        * We need to do this before we release the directory semaphore.
+        */
+       if (dir == dir->i_sb->s_root->d_inode)
+               dentry->d_op = &autofs4_root_dentry_operations;
+       else
+               dentry->d_op = &autofs4_dentry_operations;
+
+       if (!oz_mode)
+               dentry->d_flags |= DCACHE_AUTOFS_PENDING;
+       dentry->d_fsdata = NULL;
+       d_add(dentry, NULL);
+
+       if (dentry->d_op && dentry->d_op->d_revalidate) {
+               up(&dir->i_sem);
+               (dentry->d_op->d_revalidate)(dentry, 0);
+               down(&dir->i_sem);
+       }
+
+       /*
+        * If we are still pending, check if we had to handle
+        * a signal. If so we can force a restart..
+        */
+       if (dentry->d_flags & DCACHE_AUTOFS_PENDING) {
+               if (signal_pending(current))
+                       return ERR_PTR(-ERESTARTNOINTR);
+       }
+
+       /*
+        * If this dentry is unhashed, then we shouldn't honour this
+        * lookup even if the dentry is positive.  Returning ENOENT here
+        * doesn't do the right thing for all system calls, but it should
+        * be OK for the operations we permit from an autofs.
+        */
+       if ( dentry->d_inode && list_empty(&dentry->d_hash) )
+               return ERR_PTR(-ENOENT);
+
+       return NULL;
+}
+
+static int autofs4_dir_symlink(struct inode *dir,
+                               struct dentry *dentry,
+                               const char *symname)
+{
+       struct autofs_sb_info *sbi = autofs4_sbi(dir->i_sb);
+       struct autofs_info *ino = autofs4_dentry_ino(dentry);
+       struct inode *inode;
+       char *cp;
+
+       DPRINTK(("autofs_dir_symlink: %s <- %.*s\n", symname,
+                dentry->d_name.len, dentry->d_name.name));
```

```
+
+       if (!S_ISDIR(dir->i_mode))
+               return -ENOTDIR;
+
+       if (!autofs4_oz_mode(sbi))
+               return -EACCES;
+
+       if (dentry->d_name.len > NAME_MAX)
+               return -ENAMETOOLONG;
+
+       if (dentry->d_inode != NULL)
+               return -EEXIST;
+
+       ino = autofs4_init_ino(ino, sbi, S_IFLNK | 0555);
+       if (ino == NULL)
+               return -ENOSPC;
+
+       ino->size = strlen(symname);
+       ino->u.symlink = cp = kmalloc(ino->size + 1, GFP_KERNEL);
+
+       if (cp == NULL) {
+               kfree(ino);
+               return -ENOSPC;
+       }
+
+       strcpy(cp, symname);
+
+       inode = autofs4_get_inode(dir->i_sb, ino);
+       d_instantiate(dentry, inode);
+
+       if (dir == dir->i_sb->s_root->d_inode)
+               dentry->d_op = &autofs4_root_dentry_operations;
+       else
+               dentry->d_op = &autofs4_dentry_operations;
+
+       dentry->d_fsdata = ino;
+       ino->dentry = dget(dentry);
+       ino->inode = inode;
+
+       dir->i_mtime = CURRENT_TIME;
+
+       return 0;
+}
+
+/*
+ * NOTE!
+ *
+ * Normal filesystems would do a "d_delete()" to tell the VFS dcache
+ * that the file no longer exists. However, doing that means that the
+ * VFS layer can turn the dentry into a negative dentry.  We don't want
+ * this, because since the unlink is probably the result of an expire.
+ * We simply d_drop it, which allows the dentry lookup to remount it
+ * if necessary.
+ *
+ * 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()..
+ */
+static int autofs4_dir_unlink(struct inode *dir, struct dentry *dentry)
```

```
+{
+       struct autofs_sb_info *sbi = autofs4_sbi(dir->i_sb);
+       struct autofs_info *ino = autofs4_dentry_ino(dentry);
+
+       if (!S_ISDIR(dir->i_mode))
+               return -ENOTDIR;
+
+       if (dentry->d_inode == NULL)
+               return -ENOENT;
+
+       /* This allows root to remove symlinks */
+       if ( !autofs4_oz_mode(sbi) && !capable(CAP_SYS_ADMIN) )
+               return -EACCES;
+
+       dput(ino->dentry);
+
+       dentry->d_inode->i_size = 0;
+       dentry->d_inode->i_nlink = 0;
+
+       dir->i_mtime = CURRENT_TIME;
+
+       DPRINTK(("autofs_dir_unlink: unlinking %p %.*s, count=%d\n",
+               dentry, dentry->d_name.len, dentry->d_name.name, dentry->d_count));
+
+       d_drop(dentry);
+
+       return 0;
+}
+
+static int autofs4_dir_rmdir(struct inode *dir, struct dentry *dentry)
+{
+       struct autofs_sb_info *sbi = autofs4_sbi(dir->i_sb);
+       struct autofs_info *ino = autofs4_dentry_ino(dentry);
+
+       if (!S_ISDIR(dir->i_mode))
+               return -ENOTDIR;
+
+       if (dentry->d_inode == NULL)
+               return -ENOENT;
+
+       if (!autofs4_oz_mode(sbi))
+               return -EACCES;
+
+       if (!list_empty(&dentry->d_subdirs))
+               return -ENOTEMPTY;
+       dput(ino->dentry);
+
+       dentry->d_inode->i_size = 0;
+       dentry->d_inode->i_nlink = 0;
+
+       if (dir->i_nlink)
+               dir->i_nlink--;
+
+       DPRINTK(("autofs_dir_rmdir: rmdir %p %.*s, count=%d\n",
+               dentry, dentry->d_name.len, dentry->d_name.name, dentry->d_count));
+
+       d_drop(dentry);
+
+       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);
+       struct autofs_info *ino = autofs4_dentry_ino(dentry);
+       struct inode *inode;
+
+       if (!S_ISDIR(dir->i_mode))
+               return -ENOTDIR;
+
+       if ( !autofs4_oz_mode(sbi) )
+               return -EACCES;
+
+       if ( dentry->d_inode != NULL )
+               return -EEXIST;
+
+       if ( dentry->d_name.len > NAME_MAX )
+               return -ENAMETOOLONG;
+
+       DPRINTK(("autofs_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);
+       if (ino == NULL)
+               return -ENOSPC;
+
+       inode = autofs4_get_inode(dir->i_sb, ino);
+       d_instantiate(dentry, inode);
+
+       if (dir == dir->i_sb->s_root->d_inode)
+               dentry->d_op = &autofs4_root_dentry_operations;
+       else
+               dentry->d_op = &autofs4_dentry_operations;
+
+       dentry->d_fsdata = ino;
+       ino->dentry = dget(dentry);
+       ino->inode = inode;
+       dir->i_nlink++;
+       dir->i_mtime = CURRENT_TIME;
+
+       return 0;
+}
+
+/* Get/set timeout ioctl() operation */
+static inline int autofs4_get_set_timeout(struct autofs_sb_info *sbi,
+                                         unsigned long *p)
+{
+       int rv;
+       unsigned long ntimeout;
+
+       if ( (rv = get_user(ntimeout, p)) ||
+            (rv = put_user(sbi->exp_timeout/HZ, p)) )
+               return rv;
+
+       if ( ntimeout > ULONG_MAX/HZ )
+               sbi->exp_timeout = 0;
+       else
+               sbi->exp_timeout = ntimeout * HZ;
```

```
+
+       return 0;
+}
+
+/* Return protocol version */
+static inline int autofs4_get_protover(struct autofs_sb_info *sbi, int *p)
+{
+       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 dentry && dentry->d_inode &&
+               (dentry->d_op == &autofs4_root_dentry_operations ||
+                dentry->d_op == &autofs4_dentry_operations) &&
+               dentry->d_fsdata != NULL;
+}
+
+/*
+ * ioctl()'s on the root directory is the chief method for the daemon to
+ * generate kernel reactions
+ */
+static int autofs4_root_ioctl(struct inode *inode, struct file *filp,
+                             unsigned int cmd, unsigned long arg)
+{
+       struct autofs_sb_info *sbi = autofs4_sbi(inode->i_sb);
+
+       DPRINTK(("autofs_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) ||
+            _IOC_NR(cmd) - _IOC_NR(AUTOFS_IOC_FIRST) >= AUTOFS_IOC_COUNT )
+               return -ENOTTY;
+
+       if ( !autofs4_oz_mode(sbi) && !capable(CAP_SYS_ADMIN) )
+               return -EPERM;
+
+       switch(cmd) {
+       case AUTOFS_IOC_READY:  /* Wait queue: go ahead and retry */
+               return autofs4_wait_release(sbi,(autofs_wqt_t)arg,0);
+       case AUTOFS_IOC_FAIL:   /* Wait queue: fail with ENOENT */
+               return autofs4_wait_release(sbi,(autofs_wqt_t)arg,-ENOENT);
+       case AUTOFS_IOC_CATATONIC: /* Enter catatonic mode (daemon shutdown) */
+               autofs4_catatonic_mode(sbi);
+               return 0;
+       case AUTOFS_IOC_PROTOVER: /* Get protocol version */
+               return autofs4_get_protover(sbi, (int *)arg);
+       case AUTOFS_IOC_SETTIMEOUT:
+               return autofs4_get_set_timeout(sbi,(unsigned long *)arg);
+
+       /* return a single thing to expire */
+       case AUTOFS_IOC_EXPIRE:
+               return autofs4_expire_run(inode->i_sb,sbi,
+                                         (struct autofs_packet_expire *)arg);
+       /* same as above, but can send multiple expires through pipe */
+       case AUTOFS_IOC_EXPIRE_MULTI:
```

```
+               return autofs4_expire_multi(inode->i_sb, sbi, (int *)arg);
+
+       default:
+               return -ENOSYS;
+       }
+}
diff -X diffexcl -Nur 2.2/fs/autofs4/symlink.c autofs-2.2/fs/autofs4/symlink.c
--- 2.2/fs/autofs4/symlink.c    Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/symlink.c  Sun Oct 22 19:47:15 2000
@@ -0,0 +1,52 @@
+/* -*- c -*- --------------------------------------------------------------- *
+ *
+ * linux/fs/autofs/symlink.c
+ *
+ *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ *
+ * ------------------------------------------------------------------------- */
+
+#include "autofs_i.h"
+
+static inline int vfs_readlink(struct dentry *dentry, char *buffer, int buflen,
+                               const char *link)
+{
+       int len;
+
+       len = PTR_ERR(link);
+       if (IS_ERR(link))
+               goto out;
+
+       len = strlen(link);
+       if (len > (unsigned) buflen)
+               len = buflen;
+       if (copy_to_user(buffer, link, len))
+               len = -EFAULT;
+out:
+       return len;
+}
+
+static int autofs4_readlink(struct dentry *dentry, char *buffer, int buflen)
+{
+       struct autofs_info *ino = autofs4_dentry_ino(dentry);
+
+       return vfs_readlink(dentry, buffer, buflen, ino->u.symlink);
+}
+
+static struct dentry * autofs4_follow_link(struct dentry *dentry,
+                                          struct dentry *base,
+                                          unsigned int flags)
+{
+       struct autofs_info *ino = autofs4_dentry_ino(dentry);
+
+       return lookup_dentry(ino->u.symlink, base, flags);
+}
+
+struct inode_operations autofs4_symlink_inode_operations = {
+       readlink:       autofs4_readlink,
```

```
+       follow_link:    autofs4_follow_link
+};
diff -X diffexcl -Nur 2.2/fs/autofs4/waitq.c autofs-2.2/fs/autofs4/waitq.c
--- 2.2/fs/autofs4/waitq.c      Wed Dec 31 16:00:00 1969
+++ linux/fs/autofs4/waitq.c    Mon Oct 23 14:17:58 2000
@@ -0,0 +1,250 @@
+/* -*- c -*- --------------------------------------------------------------- *
+ *
+ * linux/fs/autofs/waitq.c
+ *
+ *  Copyright 1997-1998 Transmeta Corporation -- All Rights Reserved
+ *
+ * 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
+ * option, any later version, incorporated herein by reference.
+ *
+ * ------------------------------------------------------------------------- */
+
+#include <linux/malloc.h>
+#include <linux/sched.h>
+#include <linux/signal.h>
+#include <linux/file.h>
+#include "autofs_i.h"
+
+/* We make this a static variable rather than a part of the superblock; it
+   is better if we don't reassign numbers easily even across filesystems */
+static autofs_wqt_t autofs4_next_wait_queue = 1;
+
+/* These are the signals we allow interrupting a pending mount */
+#define SHUTDOWN_SIGS  (sigmask(SIGKILL) | sigmask(SIGINT) | sigmask(SIGQUIT))
+
+void autofs4_catatonic_mode(struct autofs_sb_info *sbi)
+{
+       struct autofs_wait_queue *wq, *nwq;
+
+       DPRINTK(("autofs: entering catatonic mode\n"));
+
+       sbi->catatonic = 1;
+       wq = sbi->queues;
+       sbi->queues = NULL;     /* Erase all wait queues */
+       while ( wq ) {
+               nwq = wq->next;
+               wq->status = -ENOENT; /* Magic is gone - report failure */
+               kfree(wq->name);
+               wq->name = NULL;
+               wake_up(&wq->queue);
+               wq = nwq;
+       }
+       if (sbi->pipe) {
+               fput(sbi->pipe);        /* Close the pipe */
+               sbi->pipe = NULL;
+       }
+
+       shrink_dcache_sb(sbi->sb);
+}
+
+static int autofs4_write(struct file *file, const void *addr, int bytes)
+{
+       unsigned long sigpipe, flags;
+       mm_segment_t fs;
```

```
+       const char *data = (const char *)addr;
+       ssize_t wr = 0;
+
+       /** WARNING: this is not safe for writing more than PIPE_BUF bytes! **/
+
+       sigpipe = sigismember(&current->signal, SIGPIPE);
+
+       /* Save pointer to user space and point back to kernel space */
+       fs = get_fs();
+       set_fs(KERNEL_DS);
+
+       while (bytes &&
+              (wr = file->f_op->write(file,data,bytes,&file->f_pos)) > 0) {
+               data += wr;
+               bytes -= wr;
+       }
+
+       set_fs(fs);
+
+       /* Keep the currently executing process from receiving a
+          SIGPIPE unless it was already supposed to get one */
+       if (wr == -EPIPE && !sigpipe) {
+               spin_lock_irqsave(&current->sigmask_lock, flags);
+               sigdelset(&current->signal, SIGPIPE);
+               recalc_sigpending(current);
+               spin_unlock_irqrestore(&current->sigmask_lock, flags);
+       }
+
+       return (bytes > 0);
+}
+
+static void autofs4_notify_daemon(struct autofs_sb_info *sbi,
+                                 struct autofs_wait_queue *wq,
+                                 int type)
+{
+       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));
+
+       memset(&pkt,0,sizeof pkt); /* For security reasons */
+
+       pkt.hdr.proto_version = sbi->version;
+       pkt.hdr.type = type;
+       if (type == autofs_ptype_missing) {
+               struct autofs_packet_missing *mp = &pkt.missing;
+
+               pktsz = sizeof(*mp);
+
+               mp->wait_queue_token = wq->wait_queue_token;
+               mp->len = wq->len;
+               memcpy(mp->name, wq->name, wq->len);
+               mp->name[wq->len] = '\0';
+       } else if (type == autofs_ptype_expire_multi) {
+               struct autofs_packet_expire_multi *ep = &pkt.expire_multi;
+
+               pktsz = sizeof(*ep);
+
+               ep->wait_queue_token = wq->wait_queue_token;
```

```
+               ep->len = wq->len;
+               memcpy(ep->name, wq->name, wq->len);
+               ep->name[wq->len] = '\0';
+       } else {
+               printk("autofs_notify_daemon: bad type %d!\n", type);
+               return;
+       }
+
+       if (autofs4_write(sbi->pipe, &pkt, pktsz))
+               autofs4_catatonic_mode(sbi);
+}
+
+int autofs4_wait(struct autofs_sb_info *sbi, struct qstr *name,
+               enum autofs_notify notify)
+{
+       struct autofs_wait_queue *wq;
+       int status;
+
+       /* In catatonic mode, we don't wait for nobody */
+       if ( sbi->catatonic )
+               return -ENOENT;
+
+       /* We shouldn't be able to get here, but just in case */
+       if ( name->len > NAME_MAX )
+               return -ENOENT;
+
+       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) )
+                       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);
+                       return -ENOMEM;
+               }
+               wq->wait_queue_token = autofs4_next_wait_queue;
+               if (++autofs4_next_wait_queue == 0)
+                       autofs4_next_wait_queue = 1;
+               init_waitqueue(&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;
+
+               DPRINTK(("autofs_wait: new wait id = 0x%08lx, name = %.*s, nfy=%d\n",
+                        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)
+               }
+       } 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));
+       }
+
+       /* wq->name is NULL if and only if the lock is already released */
+
+       if ( sbi->catatonic ) {
+               /* We might have slept, so check again for catatonic mode */
+               wq->status = -ENOENT;
+               if ( wq->name ) {
+                       kfree(wq->name);
+                       wq->name = NULL;
+               }
+       }
+
+       if ( wq->name ) {
+               /* Block all but "shutdown" signals while waiting */
+               sigset_t oldset;
+               unsigned long irqflags;
+
+               spin_lock_irqsave(&current->sigmask_lock, irqflags);
+               oldset = current->blocked;
+               siginitsetinv(&current->blocked, SHUTDOWN_SIGS & ~oldset.sig[0]);
+               recalc_sigpending(current);
+               spin_unlock_irqrestore(&current->sigmask_lock, irqflags);
+
+               interruptible_sleep_on(&wq->queue);
+
+               spin_lock_irqsave(&current->sigmask_lock, irqflags);
+               current->blocked = oldset;
+               recalc_sigpending(current);
+               spin_unlock_irqrestore(&current->sigmask_lock, irqflags);
+       } else {
+               DPRINTK(("autofs_wait: skipped sleeping\n"));
+       }
+
+       status = wq->status;
+
+       if (--wq->wait_ctr == 0)        /* Are we the last process to need status? */
+               kfree(wq);
+
+       return status;
+}
+
+
+int autofs4_wait_release(struct autofs_sb_info *sbi, autofs_wqt_t wait_queue_token, int statu
+{
+       struct autofs_wait_queue *wq, **wql;
+
+       for ( wql = &sbi->queues ; (wq = *wql) ; wql = &wq->next ) {
+               if ( wq->wait_queue_token == wait_queue_token )
+                       break;
+       }
+       if ( !wq )
```

```
+               return -EINVAL;
+
+       *wql = wq->next;        /* Unlink from chain */
+       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? */
+               kfree(wq);
+       else
+               wake_up(&wq->queue);
+
+       return 0;
+}
+
```