```
**********************************************************
  121852 Mon Sep 28 19:41:48 2015
new/usr/src/uts/common/fs/vfs.c
6265 speed up mount/umount
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 */

/*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
/*        All Rights Reserved   */

/*
 * University Copyright- Copyright (c) 1982, 1986, 1988
 * The Regents of the University of California
 * All Rights Reserved
 *
 * University Acknowledgment- Portions of this document are derived from
 * software developed by the University of California, Berkeley, and its
 * contributors.
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/errno.h>
#include <sys/user.h>
#include <sys/fstyp.h>
#include <sys/kmem.h>
#include <sys/systm.h>
#include <sys/proc.h>
#include <sys/mount.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/fem.h>
#include <sys/mntent.h>
#include <sys/stat.h>
#include <sys/statvfs.h>
#include <sys/statfs.h>
#include <sys/cred.h>
#include <sys/vnode.h>
#include <sys/rwstlock.h>
#include <sys/dnlc.h>
#include <sys/file.h>
#include <sys/time.h>
```

```
#include <sys/atomic.h>
#include <sys/cmn_err.h>
#include <sys/buf.h>
#include <sys/swap.h>
#include <sys/debug.h>
#include <sys/vnode.h>
#include <sys/modctl.h>
#include <sys/ddi.h>
#include <sys/pathname.h>
#include <sys/bootconf.h>
#include <sys/dumphdr.h>
#include <sys/dc_ki.h>
#include <sys/poll.h>
#include <sys/sunddi.h>
#include <sys/sysmacros.h>
#include <sys/zone.h>
#include <sys/policy.h>
#include <sys/ctfs.h>
#include <sys/objfs.h>
#include <sys/console.h>
#include <sys/reboot.h>
#include <sys/attr.h>
#include <sys/zio.h>
#include <sys/spa.h>
#include <sys/lofi.h>
#include <sys/bootprops.h>
#include <sys/avl.h>
#endif /* ! codereview */

#include <vm/page.h>

#include <fs/fs_subr.h>
/* Private interfaces to create vopstats-related data structures */
extern void             initialize_vopstats(vopstats_t *);
extern vopstats_t       *get_fstype_vopstats(struct vfs *, struct vfssw *);
extern vsk_anchor_t     *get_vskstat_anchor(struct vfs *);

static void vfs_clearmntopt_nolock(mntopts_t *, const char *, int);
static void vfs_setmntopt_nolock(mntopts_t *, const char *,
    const char *, int, int);
static int  vfs_optionisset_nolock(const mntopts_t *, const char *, char **);
static void vfs_freemnttab(struct vfs *);
static void vfs_freeopt(mntopt_t *);
static void vfs_swapopttbl_nolock(mntopts_t *, mntopts_t *);
static void vfs_swapopttbl(mntopts_t *, mntopts_t *);
static void vfs_copyopttbl_extend(const mntopts_t *, mntopts_t *, int);
static void vfs_createopttbl_extend(mntopts_t *, const char *,
    const mntopts_t *);
static char **vfs_copycancelopt_extend(char **const, int);
static void vfs_freecancelopt(char **);
static void getrootfs(char **, char **);
static int getmacpath(dev_info_t *, void *);
static void vfs_mnttabvp_setup(void);

struct ipmnt {
        struct ipmnt    *mip_next;
        dev_t           mip_dev;
        struct vfs      *mip_vfsp;
};

static kmutex_t         vfs_miplist_mutex;
static struct ipmnt     *vfs_miplist = NULL;
static struct ipmnt     *vfs_miplist_end = NULL;

static kmem_cache_t *vfs_cache; /* Pointer to VFS kmem cache */
```

```
/*
 * VFS global data.
 */
vnode_t *rootdir;               /* pointer to root inode vnode. */
vnode_t *devicesdir;            /* pointer to inode of devices root */
vnode_t *devdir;                /* pointer to inode of dev root */

char *server_rootpath;          /* root path for diskless clients */
char *server_hostname;          /* hostname of diskless server */

static struct vfs root;
static struct vfs devices;
static struct vfs dev;
struct vfs *rootvfs = &root;    /* pointer to root vfs; head of VFS list. */
avl_tree_t vfs_by_dev;          /* avl tree to index mounted VFSs by dev */
avl_tree_t vfs_by_mntpnt;       /* avl tree to index mounted VFSs by mntpnt */
uint64_t vfs_curr_mntix;        /* counter to provide a unique mntix for
                                 * entries in the above avl trees.
                                 * protected by vfslist lock */
#endif /* ! codereview */
rvfs_t *rvfs_list;              /* array of vfs ptrs for vfs hash list */
int vfshsz = 512;               /* # of heads/locks in vfs hash arrays */
                                /* must be power of 2!  */
timespec_t vfs_mnttab_ctime;    /* mnttab created time */
timespec_t vfs_mnttab_mtime;    /* mnttab last modified time */
char *vfs_dummyfstype = "\0";
struct pollhead vfs_pollhd;     /* for mnttab pollers */
struct vnode *vfs_mntdummyvp;   /* to fake mnttab read/write for file events */
int     mntfstype;              /* will be set once mnt fs is mounted */

/*
 * Table for generic options recognized in the VFS layer and acted
 * on at this level before parsing file system specific options.
 * The nosuid option is stronger than any of the devices and setuid
 * options, so those are canceled when nosuid is seen.
 *
 * All options which are added here need to be added to the
 * list of standard options in usr/src/cmd/fs.d/fslib.c as well.
 */
/*
 * VFS Mount options table
 */
static char *ro_cancel[] = { MNTOPT_RW, NULL };
static char *rw_cancel[] = { MNTOPT_RO, NULL };
static char *suid_cancel[] = { MNTOPT_NOSUID, NULL };
static char *nosuid_cancel[] = { MNTOPT_SUID, MNTOPT_DEVICES, MNTOPT_NODEVICES,
    MNTOPT_NOSETUID, MNTOPT_SETUID, NULL };
static char *devices_cancel[] = { MNTOPT_NODEVICES, NULL };
static char *nodevices_cancel[] = { MNTOPT_DEVICES, NULL };
static char *setuid_cancel[] = { MNTOPT_NOSETUID, NULL };
static char *nosetuid_cancel[] = { MNTOPT_SETUID, NULL };
static char *nbmand_cancel[] = { MNTOPT_NONBMAND, NULL };
static char *nonbmand_cancel[] = { MNTOPT_NBMAND, NULL };
static char *exec_cancel[] = { MNTOPT_NOEXEC, NULL };
static char *noexec_cancel[] = { MNTOPT_EXEC, NULL };

static const mntopt_t mntopts[] = {
/*
 *      option name             cancel options          default arg     flags
 */
        { MNTOPT_REMOUNT,       NULL,                   NULL,
                MO_NODISPLAY, (void *)0 },
        { MNTOPT_RO,            ro_cancel,              NULL,           0,
                (void *)0 },
        { MNTOPT_RW,            rw_cancel,              NULL,           0,
                (void *)0 },
```

```
        { MNTOPT_SUID,          suid_cancel,            NULL,           0,
                (void *)0 },
        { MNTOPT_NOSUID,        nosuid_cancel,          NULL,           0,
                (void *)0 },
        { MNTOPT_DEVICES,       devices_cancel,         NULL,           0,
                (void *)0 },
        { MNTOPT_NODEVICES,     nodevices_cancel,       NULL,           0,
                (void *)0 },
        { MNTOPT_SETUID,        setuid_cancel,          NULL,           0,
                (void *)0 },
        { MNTOPT_NOSETUID,      nosetuid_cancel,        NULL,           0,
                (void *)0 },
        { MNTOPT_NBMAND,        nbmand_cancel,          NULL,           0,
                (void *)0 },
        { MNTOPT_NONBMAND,      nonbmand_cancel,        NULL,           0,
                (void *)0 },
        { MNTOPT_EXEC,          exec_cancel,            NULL,           0,
                (void *)0 },
        { MNTOPT_NOEXEC,        noexec_cancel,          NULL,           0,
                (void *)0 },
};

const mntopts_t vfs_mntopts = {
        sizeof (mntopts) / sizeof (mntopt_t),
        (mntopt_t *)&mntopts[0]
};

/*
 * File system operation dispatch functions.
 */

int
fsop_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
{
        return (*(vfsp)->vfs_op->vfs_mount)(vfsp, mvp, uap, cr);
}

int
fsop_unmount(vfs_t *vfsp, int flag, cred_t *cr)
{
        return (*(vfsp)->vfs_op->vfs_unmount)(vfsp, flag, cr);
}

int
fsop_root(vfs_t *vfsp, vnode_t **vpp)
{
        refstr_t *mntpt;
        int ret = (*(vfsp)->vfs_op->vfs_root)(vfsp, vpp);
        /*
         * Make sure this root has a path.  With lofs, it is possible to have
         * a NULL mountpoint.
         */
        if (ret == 0 && vfsp->vfs_mntpt != NULL && (*vpp)->v_path == NULL) {
                mntpt = vfs_getmntpoint(vfsp);
                vn_setpath_str(*vpp, refstr_value(mntpt),
                    strlen(refstr_value(mntpt)));
                refstr_rele(mntpt);
        }

        return (ret);
}

int
fsop_statfs(vfs_t *vfsp, statvfs64_t *sp)
{
        return (*(vfsp)->vfs_op->vfs_statvfs)(vfsp, sp);
```

```c
}

int
fsop_sync(vfs_t *vfsp, short flag, cred_t *cr)
{
        return (*(vfsp)->vfs_op->vfs_sync)(vfsp, flag, cr);
}

int
fsop_vget(vfs_t *vfsp, vnode_t **vpp, fid_t *fidp)
{
        /*
         * In order to handle system attribute fids in a manner
         * transparent to the underlying fs, we embed the fid for
         * the sysattr parent object in the sysattr fid and tack on
         * some extra bytes that only the sysattr layer knows about.
         *
         * This guarantees that sysattr fids are larger than other fids
         * for this vfs. If the vfs supports the sysattr view interface
         * (as indicated by VFSFT_SYSATTR_VIEWS), we cannot have a size
         * collision with XATTR_FIDSZ.
         */
        if (vfs_has_feature(vfsp, VFSFT_SYSATTR_VIEWS) &&
            fidp->fid_len == XATTR_FIDSZ)
                return (xattr_dir_vget(vfsp, vpp, fidp));

        return (*(vfsp)->vfs_op->vfs_vget)(vfsp, vpp, fidp);
}

int
fsop_mountroot(vfs_t *vfsp, enum whymountroot reason)
{
        return (*(vfsp)->vfs_op->vfs_mountroot)(vfsp, reason);
}

void
fsop_freefs(vfs_t *vfsp)
{
        (*(vfsp)->vfs_op->vfs_freevfs)(vfsp);
}

int
fsop_vnstate(vfs_t *vfsp, vnode_t *vp, vntrans_t nstate)
{
        return ((*(vfsp)->vfs_op->vfs_vnstate)(vfsp, vp, nstate));
}

int
fsop_sync_by_kind(int fstype, short flag, cred_t *cr)
{
        ASSERT((fstype >= 0) && (fstype < nfstype));

        if (ALLOCATED_VFSSW(&vfssw[fstype]) && VFS_INSTALLED(&vfssw[fstype]))
                return (*vfssw[fstype].vsw_vfsops.vfs_sync) (NULL, flag, cr);
        else
                return (ENOTSUP);
}

/*
 * File system initialization.  vfs_setfsops() must be called from a file
 * system's init routine.
 */

static int
fs_copyfsops(const fs_operation_def_t *template, vfsops_t *actual,
    int *unused_ops)
```

```c
{
        static const fs_operation_trans_def_t vfs_ops_table[] = {
                VFSNAME_MOUNT, offsetof(vfsops_t, vfs_mount),
                        fs_nosys, fs_nosys,

                VFSNAME_UNMOUNT, offsetof(vfsops_t, vfs_unmount),
                        fs_nosys, fs_nosys,

                VFSNAME_ROOT, offsetof(vfsops_t, vfs_root),
                        fs_nosys, fs_nosys,

                VFSNAME_STATVFS, offsetof(vfsops_t, vfs_statvfs),
                        fs_nosys, fs_nosys,

                VFSNAME_SYNC, offsetof(vfsops_t, vfs_sync),
                        (fs_generic_func_p) fs_sync,
                        (fs_generic_func_p) fs_sync,    /* No errors allowed */

                VFSNAME_VGET, offsetof(vfsops_t, vfs_vget),
                        fs_nosys, fs_nosys,

                VFSNAME_MOUNTROOT, offsetof(vfsops_t, vfs_mountroot),
                        fs_nosys, fs_nosys,

                VFSNAME_FREEVFS, offsetof(vfsops_t, vfs_freevfs),
                        (fs_generic_func_p)fs_freevfs,
                        (fs_generic_func_p)fs_freevfs,  /* Shouldn't fail */

                VFSNAME_VNSTATE, offsetof(vfsops_t, vfs_vnstate),
                        (fs_generic_func_p)fs_nosys,
                        (fs_generic_func_p)fs_nosys,

                NULL, 0, NULL, NULL
        };

        return (fs_build_vector(actual, unused_ops, vfs_ops_table, template));
}

void
zfs_boot_init() {

        if (strcmp(rootfs.bo_fstype, MNTTYPE_ZFS) == 0)
                spa_boot_init();
}

int
vfs_setfsops(int fstype, const fs_operation_def_t *template, vfsops_t **actual)
{
        int error;
        int unused_ops;

        /*
         * Verify that fstype refers to a valid fs.  Note that
         * 0 is valid since it's used to set "stray" ops.
         */
        if ((fstype < 0) || (fstype >= nfstype))
                return (EINVAL);

        if (!ALLOCATED_VFSSW(&vfssw[fstype]))
                return (EINVAL);

        /* Set up the operations vector. */

        error = fs_copyfsops(template, &vfssw[fstype].vsw_vfsops, &unused_ops);

        if (error != 0)
```

```
                return (error);

        vfssw[fstype].vsw_flag |= VSW_INSTALLED;

        if (actual != NULL)
                *actual = &vfssw[fstype].vsw_vfsops;

#if DEBUG
        if (unused_ops != 0)
                cmn_err(CE_WARN, "vfs_setfsops: %s: %d operations supplied "
                    "but not used", vfssw[fstype].vsw_name, unused_ops);
#endif

        return (0);
}

int
vfs_makefsops(const fs_operation_def_t *template, vfsops_t **actual)
{
        int error;
        int unused_ops;

        *actual = (vfsops_t *)kmem_alloc(sizeof (vfsops_t), KM_SLEEP);

        error = fs_copyfsops(template, *actual, &unused_ops);
        if (error != 0) {
                kmem_free(*actual, sizeof (vfsops_t));
                *actual = NULL;
                return (error);
        }

        return (0);
}

/*
 * Free a vfsops structure created as a result of vfs_makefsops().
 * NOTE: For a vfsops structure initialized by vfs_setfsops(), use
 * vfs_freevfsops_by_type().
 */
void
vfs_freevfsops(vfsops_t *vfsops)
{
        kmem_free(vfsops, sizeof (vfsops_t));
}

/*
 * Since the vfsops structure is part of the vfssw table and wasn’t
 * really allocated, we’re not really freeing anything.  We keep
 * the name for consistency with vfs_freevfsops().  We do, however,
 * need to take care of a little bookkeeping.
 * NOTE: For a vfsops structure created by vfs_setfsops(), use
 * vfs_freevfsops_by_type().
 */
int
vfs_freevfsops_by_type(int fstype)
{

        /* Verify that fstype refers to a loaded fs (and not fsid 0). */
        if ((fstype <= 0) || (fstype >= nfstype))
                return (EINVAL);

        WLOCK_VFSSW();
        if ((vfssw[fstype].vsw_flag & VSW_INSTALLED) == 0) {
                WUNLOCK_VFSSW();
                return (EINVAL);
        }
```

```

        vfssw[fstype].vsw_flag &= ~VSW_INSTALLED;
        WUNLOCK_VFSSW();

        return (0);
}

/* Support routines used to reference vfs_op */

/* Set the operations vector for a vfs */
void
vfs_setops(vfs_t *vfsp, vfsops_t *vfsops)
{
        vfsops_t        *op;

        ASSERT(vfsp != NULL);
        ASSERT(vfsops != NULL);

        op = vfsp->vfs_op;
        membar_consumer();
        if (vfsp->vfs_femhead == NULL &&
            atomic_cas_ptr(&vfsp->vfs_op, op, vfsops) == op) {
                return;
        }
        fsem_setvfsops(vfsp, vfsops);
}

/* Retrieve the operations vector for a vfs */
vfsops_t *
vfs_getops(vfs_t *vfsp)
{
        vfsops_t        *op;

        ASSERT(vfsp != NULL);

        op = vfsp->vfs_op;
        membar_consumer();
        if (vfsp->vfs_femhead == NULL && op == vfsp->vfs_op) {
                return (op);
        } else {
                return (fsem_getvfsops(vfsp));
        }
}

/*
 * Returns non-zero (1) if the vfsops matches that of the vfs.
 * Returns zero (0) if not.
 */
int
vfs_matchops(vfs_t *vfsp, vfsops_t *vfsops)
{
        return (vfs_getops(vfsp) == vfsops);
}

/*
 * Returns non-zero (1) if the file system has installed a non-default,
 * non-error vfs_sync routine.  Returns zero (0) otherwise.
 */
int
vfs_can_sync(vfs_t *vfsp)
{
        /* vfs_sync() routine is not the default/error function */
        return (vfs_getops(vfsp)->vfs_sync != fs_sync);
}

/*
```

```c
 * Initialize a vfs structure.
 */
void
vfs_init(vfs_t *vfsp, vfsops_t *op, void *data)
{
        /* Other initialization has been moved to vfs_alloc() */
        vfsp->vfs_count = 0;
        vfsp->vfs_next = vfsp;
        vfsp->vfs_prev = vfsp;
        vfsp->vfs_zone_next = vfsp;
        vfsp->vfs_zone_prev = vfsp;
        vfsp->vfs_lofi_minor = 0;
        sema_init(&vfsp->vfs_reflock, 1, NULL, SEMA_DEFAULT, NULL);
        vfsimpl_setup(vfsp);
        vfsp->vfs_data = (data);
        vfs_setops((vfsp), (op));
}

/*
 * Allocate and initialize the vfs implementation private data
 * structure, vfs_impl_t.
 */
void
vfsimpl_setup(vfs_t *vfsp)
{
        int i;

        if (vfsp->vfs_implp != NULL) {
                return;
        }

        vfsp->vfs_implp = kmem_alloc(sizeof (vfs_impl_t), KM_SLEEP);
        /* Note that these are #define'd in vfs.h */
        vfsp->vfs_vskap = NULL;
        vfsp->vfs_fstypevsp = NULL;

        /* Set size of counted array, then zero the array */
        vfsp->vfs_featureset[0] = VFS_FEATURE_MAXSZ - 1;
        for (i = 1; i <  VFS_FEATURE_MAXSZ; i++) {
                vfsp->vfs_featureset[i] = 0;
        }
}

/*
 * Release the vfs_impl_t structure, if it exists. Some unbundled
 * filesystems may not use the newer version of vfs and thus
 * would not contain this implementation private data structure.
 */
void
vfsimpl_teardown(vfs_t *vfsp)
{
        vfs_impl_t      *vip = vfsp->vfs_implp;

        if (vip == NULL)
                return;

        kmem_free(vfsp->vfs_implp, sizeof (vfs_impl_t));
        vfsp->vfs_implp = NULL;
}

/*
 * VFS system calls: mount, umount, syssync, statfs, fstatfs, statvfs,
 * fstatvfs, and sysfs moved to common/syscall.
 */

/*
```

```c
 * Update every mounted file system.  We call the vfs_sync operation of
 * each file system type, passing it a NULL vfsp to indicate that all
 * mounted file systems of that type should be updated.
 */
void
vfs_sync(int flag)
{
        struct vfssw *vswp;
        RLOCK_VFSSW();
        for (vswp = &vfssw[1]; vswp < &vfssw[nfstype]; vswp++) {
                if (ALLOCATED_VFSSW(vswp) && VFS_INSTALLED(vswp)) {
                        vfs_refvfssw(vswp);
                        RUNLOCK_VFSSW();
                        (void) (*vswp->vsw_vfsops.vfs_sync)(NULL, flag,
                            CRED());
                        vfs_unrefvfssw(vswp);
                        RLOCK_VFSSW();
                }
        }
        RUNLOCK_VFSSW();
}

void
sync(void)
{
        vfs_sync(0);
}

/*
 * compare function for vfs_by_dev avl tree. compare dev first, then mntix
 */
static int
vfs_cmp_dev(const void *aa, const void *bb)
{
        const vfs_t *a = aa;
        const vfs_t *b = bb;

        if (a->vfs_dev < b->vfs_dev)
                return (-1);
        if (a->vfs_dev > b->vfs_dev)
                return (1);
        if (a->vfs_mntix < b->vfs_mntix)
                return (-1);
        if (a->vfs_mntix > b->vfs_mntix)
                return (1);
        return (0);
}

/*
 * compare function for vfs_by_mntpnt avl tree. compare mntpnt first, then mntix
 */
static int
vfs_cmp_mntpnt(const void *aa, const void *bb)
{
        const vfs_t *a = aa;
        const vfs_t *b = bb;
        int ret;

        ret = strcmp(refstr_value(a->vfs_mntpt), refstr_value(b->vfs_mntpt));
        if (ret < 0)
                return (-1);
        if (ret > 0)
                return (1);
        if (a->vfs_mntix < b->vfs_mntix)
                return (-1);
        if (a->vfs_mntix > b->vfs_mntix)
```

```
                return (1);
        return (0);
}

/*
#endif /* ! codereview */
 * External routines.
 */

krwlock_t vfssw_lock;   /* lock accesses to vfssw */

/*
 * Lock for accessing the vfs linked list.  Initialized in vfs_mountroot(),
 * but otherwise should be accessed only via vfs_list_lock() and
 * vfs_list_unlock().  Also used to protect the timestamp for mods to the list.
 */
static krwlock_t vfslist;

/*
 * Mount devfs on /devices. This is done right after root is mounted
 * to provide device access support for the system
 */
static void
vfs_mountdevices(void)
{
        struct vfssw *vsw;
        struct vnode *mvp;
        struct mounta mounta = {        /* fake mounta for devfs_mount() */
                NULL,
                NULL,
                MS_SYSSPACE,
                NULL,
                NULL,
                0,
                NULL,
                0
        };

        /*
         * _init devfs module to fill in the vfssw
         */
        if (modload("fs", "devfs") == -1)
                panic("Cannot _init devfs module");

        /*
         * Hold vfs
         */
        RLOCK_VFSSW();
        vsw = vfs_getvfsswbyname("devfs");
        VFS_INIT(&devices, &vsw->vsw_vfsops, NULL);
        VFS_HOLD(&devices);

        /*
         * Locate mount point
         */
        if (lookupname("/devices", UIO_SYSSPACE, FOLLOW, NULLVPP, &mvp))
                panic("Cannot find /devices");

        /*
         * Perform the mount of /devices
         */
        if (VFS_MOUNT(&devices, mvp, &mounta, CRED()))
                panic("Cannot mount /devices");

        RUNLOCK_VFSSW();
```

```
        /*
         * Set appropriate members and add to vfs list for mnttab display
         */
        vfs_setresource(&devices, "/devices", 0);
        vfs_setmntpoint(&devices, "/devices", 0);

        /*
         * Hold the root of /devices so it won't go away
         */
        if (VFS_ROOT(&devices, &devicesdir))
                panic("vfs_mountdevices: not devices root");

        if (vfs_lock(&devices) != 0) {
                VN_RELE(devicesdir);
                cmn_err(CE_NOTE, "Cannot acquire vfs_lock of /devices");
                return;
        }

        if (vn_vfswlock(mvp) != 0) {
                vfs_unlock(&devices);
                VN_RELE(devicesdir);
                cmn_err(CE_NOTE, "Cannot acquire vfswlock of /devices");
                return;
        }

        vfs_add(mvp, &devices, 0);
        vn_vfsunlock(mvp);
        vfs_unlock(&devices);
        VN_RELE(devicesdir);
}

/*
 * mount the first instance of /dev  to root and remain mounted
 */
static void
vfs_mountdev1(void)
{
        struct vfssw *vsw;
        struct vnode *mvp;
        struct mounta mounta = {        /* fake mounta for sdev_mount() */
                NULL,
                NULL,
                MS_SYSSPACE | MS_OVERLAY,
                NULL,
                NULL,
                0,
                NULL,
                0
        };

        /*
         * _init dev module to fill in the vfssw
         */
        if (modload("fs", "dev") == -1)
                cmn_err(CE_PANIC, "Cannot _init dev module\n");

        /*
         * Hold vfs
         */
        RLOCK_VFSSW();
        vsw = vfs_getvfsswbyname("dev");
        VFS_INIT(&dev, &vsw->vsw_vfsops, NULL);
        VFS_HOLD(&dev);

        /*
         * Locate mount point
         */
        if (lookupname("/dev", UIO_SYSSPACE, FOLLOW, NULLVPP, &mvp))
                cmn_err(CE_PANIC, "Cannot find /dev\n");

        /*
         * Perform the mount of /dev
         */
        if (VFS_MOUNT(&dev, mvp, &mounta, CRED()))
                cmn_err(CE_PANIC, "Cannot mount /dev 1\n");

        RUNLOCK_VFSSW();

        /*
         * Set appropriate members and add to vfs list for mnttab display
         */
        vfs_setresource(&dev, "/dev", 0);
        vfs_setmntpoint(&dev, "/dev", 0);

        /*
         * Hold the root of /dev so it won't go away
         */
        if (VFS_ROOT(&dev, &devdir))
                cmn_err(CE_PANIC, "vfs_mountdev1: not dev root");

        if (vfs_lock(&dev) != 0) {
                VN_RELE(devdir);
                cmn_err(CE_NOTE, "Cannot acquire vfs_lock of /dev");
                return;
        }

        if (vn_vfswlock(mvp) != 0) {
                vfs_unlock(&dev);
                VN_RELE(devdir);
                cmn_err(CE_NOTE, "Cannot acquire vfswlock of /dev");
                return;
        }

        vfs_add(mvp, &dev, 0);
        vn_vfsunlock(mvp);
        vfs_unlock(&dev);
        VN_RELE(devdir);
}

/*
 * Mount required filesystem. This is done right after root is mounted.
 */
static void
vfs_mountfs(char *module, char *spec, char *path)
{
        struct vnode *mvp;
        struct mounta mounta;
        vfs_t *vfsp;

        mounta.flags = MS_SYSSPACE | MS_DATA;
        mounta.fstype = module;
        mounta.spec = spec;
        mounta.dir = path;
        if (lookupname(path, UIO_SYSSPACE, FOLLOW, NULLVPP, &mvp)) {
                cmn_err(CE_WARN, "Cannot find %s", path);
                return;
        }
        if (domount(NULL, &mounta, mvp, CRED(), &vfsp))
                cmn_err(CE_WARN, "Cannot mount %s", path);
        else
                VFS_RELE(vfsp);
        VN_RELE(mvp);
```

```
}

/*
 * vfs_mountroot is called by main() to mount the root filesystem.
 */
void
vfs_mountroot(void)
{
        struct vnode    *rvp = NULL;
        char            *path;
        size_t          plen;
        struct vfssw    *vswp;
        proc_t          *p;

        rw_init(&vfssw_lock, NULL, RW_DEFAULT, NULL);
        rw_init(&vfslist, NULL, RW_DEFAULT, NULL);

        /*
         * Alloc the avl trees for quick indexing via dev and mountpoint
         */
        avl_create(&vfs_by_dev, vfs_cmp_dev, sizeof(vfs_t),
            offsetof(vfs_t, vfs_avldev));
        avl_create(&vfs_by_mntpnt, vfs_cmp_mntpnt, sizeof(vfs_t),
            offsetof(vfs_t, vfs_avlmntpnt));

        /*
#endif /* ! codereview */
         * Alloc the vfs hash bucket array and locks
         */
        rvfs_list = kmem_zalloc(vfshsz * sizeof (rvfs_t), KM_SLEEP);

        /*
         * Call machine-dependent routine "rootconf" to choose a root
         * file system type.
         */
        if (rootconf())
                panic("vfs_mountroot: cannot mount root");
        /*
         * Get vnode for '/'.  Set up rootdir, u.u_rdir and u.u_cdir
         * to point to it.  These are used by lookuppn() so that it
         * knows where to start from ('/' or '.').
         */
        vfs_setmntpoint(rootvfs, "/", 0);
        if (VFS_ROOT(rootvfs, &rootdir))
                panic("vfs_mountroot: no root vnode");

        /*
         * At this point, the process tree consists of p0 and possibly some
         * direct children of p0.  (i.e. there are no grandchildren)
         *
         * Walk through them all, setting their current directory.
         */
        mutex_enter(&pidlock);
        for (p = practive; p != NULL; p = p->p_next) {
                ASSERT(p == &p0 || p->p_parent == &p0);

                PTOU(p)->u_cdir = rootdir;
                VN_HOLD(PTOU(p)->u_cdir);
                PTOU(p)->u_rdir = NULL;
        }
        mutex_exit(&pidlock);

        /*
         * Setup the global zone's rootvp, now that it exists.
         */
        global_zone->zone_rootvp = rootdir;
        VN_HOLD(global_zone->zone_rootvp);

        /*
         * Notify the module code that it can begin using the
         * root filesystem instead of the boot program’s services.
         */
        modrootloaded = 1;

        /*
         * Special handling for a ZFS root file system.
         */
        zfs_boot_init();

        /*
         * Set up mnttab information for root
         */
        vfs_setresource(rootvfs, rootfs.bo_name, 0);

        /*
         * Notify cluster software that the root filesystem is available.
         */
        clboot_mountroot();

        /* Now that we’re all done with the root FS, set up its vopstats */
        if ((vswp = vfs_getvfsswbyvfsops(vfs_getops(rootvfs))) != NULL) {
                /* Set flag for statistics collection */
                if (vswp->vsw_flag & VSW_STATS) {
                        initialize_vopstats(&rootvfs->vfs_vopstats);
                        rootvfs->vfs_flag |= VFS_STATS;
                        rootvfs->vfs_fstypevsp =
                            get_fstype_vopstats(rootvfs, vswp);
                        rootvfs->vfs_vskap = get_vskstat_anchor(rootvfs);
                }
                vfs_unrefvfssw(vswp);
        }

        /*
         * Mount /devices, /dev instance 1, /system/contract, /etc/mnttab,
         * /etc/svc/volatile, /etc/dfs/sharetab, /system/object, and /proc.
         */
        vfs_mountdevices();
        vfs_mountdev1();

        vfs_mountfs("ctfs", "ctfs", CTFS_ROOT);
        vfs_mountfs("proc", "/proc", "/proc");
        vfs_mountfs("mntfs", "/etc/mnttab", "/etc/mnttab");
        vfs_mountfs("tmpfs", "/etc/svc/volatile", "/etc/svc/volatile");
        vfs_mountfs("objfs", "objfs", OBJFS_ROOT);

        if (getzoneid() == GLOBAL_ZONEID) {
                vfs_mountfs("sharefs", "sharefs", "/etc/dfs/sharetab");
        }

#ifdef __sparc
        /*
         * This bit of magic can go away when we convert sparc to
         * the new boot architecture based on ramdisk.
         *
         * Booting off a mirrored root volume:
         * At this point, we have booted and mounted root on a
         * single component of the mirror.  Complete the boot
         * by configuring SVM and converting the root to the
         * dev_t of the mirrored root device.  This dev_t conversion
         * only works because the underlying device doesn’t change.
         */
        if (root_is_svm) {
```

```
                if (svm_rootconf()) {
                        panic("vfs_mountroot: cannot remount root");
                }

                /*
                 * mnttab should reflect the new root device
                 */
                vfs_lock_wait(rootvfs);
                vfs_setresource(rootvfs, rootfs.bo_name, 0);
                vfs_unlock(rootvfs);
        }
#endif /* __sparc */

        if (strcmp(rootfs.bo_fstype, "zfs") != 0) {
                /*
                 * Look up the root device via devfs so that a dv_node is
                 * created for it. The vnode is never VN_RELE()ed.
                 * We allocate more than MAXPATHLEN so that the
                 * buffer passed to i_ddi_prompath_to_devfspath() is
                 * exactly MAXPATHLEN (the function expects a buffer
                 * of that length).
                 */
                plen = strlen("/devices");
                path = kmem_alloc(plen + MAXPATHLEN, KM_SLEEP);
                (void) strcpy(path, "/devices");

                if (i_ddi_prompath_to_devfspath(rootfs.bo_name, path + plen)
                    != DDI_SUCCESS ||
                    lookupname(path, UIO_SYSSPACE, FOLLOW, NULLVPP, &rvp)) {

                        /* NUL terminate in case "path" has garbage */
                        path[plen + MAXPATHLEN - 1] = ’\0’;
#ifdef  DEBUG
                        cmn_err(CE_WARN, "!Cannot lookup root device: %s",
                            path);
#endif
                }
                kmem_free(path, plen + MAXPATHLEN);
        }

        vfs_mnttabvp_setup();
}

/*
 * Check to see if our "block device" is actually a file.  If so,
 * automatically add a lofi device, and keep track of this fact.
 */
static int
lofi_add(const char *fsname, struct vfs *vfsp,
    mntopts_t *mntopts, struct mounta *uap)
{
        int fromspace = (uap->flags & MS_SYSSPACE) ?
            UIO_SYSSPACE : UIO_USERSPACE;
        struct lofi_ioctl *li = NULL;
        struct vnode *vp = NULL;
        struct pathname pn = { NULL };
        ldi_ident_t ldi_id;
        ldi_handle_t ldi_hdl;
        vfssw_t *vfssw;
        int minor;
        int err = 0;

        if ((vfssw = vfs_getvfssw(fsname)) == NULL)
                return (0);

        if (!(vfssw->vsw_flag & VSW_CANLOFI)) {
                vfs_unrefvfssw(vfssw);
                return (0);
        }

        vfs_unrefvfssw(vfssw);
        vfssw = NULL;

        if (pn_get(uap->spec, fromspace, &pn) != 0)
                return (0);

        if (lookupname(uap->spec, fromspace, FOLLOW, NULL, &vp) != 0)
                goto out;

        if (vp->v_type != VREG)
                goto out;

        /* OK, this is a lofi mount. */

        if ((uap->flags & (MS_REMOUNT|MS_GLOBAL)) ||
            vfs_optionisset_nolock(mntopts, MNTOPT_SUID, NULL) ||
            vfs_optionisset_nolock(mntopts, MNTOPT_SETUID, NULL) ||
            vfs_optionisset_nolock(mntopts, MNTOPT_DEVICES, NULL)) {
                err = EINVAL;
                goto out;
        }

        ldi_id = ldi_ident_from_anon();
        li = kmem_zalloc(sizeof (*li), KM_SLEEP);
        (void) strlcpy(li->li_filename, pn.pn_path, MAXPATHLEN);

        err = ldi_open_by_name("/dev/lofictl", FREAD | FWRITE, kcred,
            &ldi_hdl, ldi_id);

        if (err)
                goto out2;

        err = ldi_ioctl(ldi_hdl, LOFI_MAP_FILE, (intptr_t)li,
            FREAD | FWRITE | FKIOCTL, kcred, &minor);

        (void) ldi_close(ldi_hdl, FREAD | FWRITE, kcred);

        if (!err)
                vfsp->vfs_lofi_minor = minor;

out2:
        ldi_ident_release(ldi_id);
out:
        if (li != NULL)
                kmem_free(li, sizeof (*li));
        if (vp != NULL)
                VN_RELE(vp);
        pn_free(&pn);
        return (err);
}

static void
lofi_remove(struct vfs *vfsp)
{
        struct lofi_ioctl *li = NULL;
        ldi_ident_t ldi_id;
        ldi_handle_t ldi_hdl;
        int err;

        if (vfsp->vfs_lofi_minor == 0)
                return;
```

```
        ldi_id = ldi_ident_from_anon();

        li = kmem_zalloc(sizeof (*li), KM_SLEEP);
        li->li_minor = vfsp->vfs_lofi_minor;
        li->li_cleanup = B_TRUE;

        err = ldi_open_by_name("/dev/lofictl", FREAD | FWRITE, kcred,
            &ldi_hdl, ldi_id);

        if (err)
                goto out;

        err = ldi_ioctl(ldi_hdl, LOFI_UNMAP_FILE_MINOR, (intptr_t)li,
            FREAD | FWRITE | FKIOCTL, kcred, NULL);

        (void) ldi_close(ldi_hdl, FREAD | FWRITE, kcred);

        if (!err)
                vfsp->vfs_lofi_minor = 0;

out:
        ldi_ident_release(ldi_id);
        if (li != NULL)
                kmem_free(li, sizeof (*li));
}

/*
 * Common mount code.  Called from the system call entry point, from autofs,
 * nfsv4 trigger mounts, and from pxfs.
 *
 * Takes the effective file system type, mount arguments, the mount point
 * vnode, flags specifying whether the mount is a remount and whether it
 * should be entered into the vfs list, and credentials.  Fills in its vfspp
 * parameter with the mounted file system instance's vfs.
 *
 * Note that the effective file system type is specified as a string.  It may
 * be null, in which case it's determined from the mount arguments, and may
 * differ from the type specified in the mount arguments; this is a hook to
 * allow interposition when instantiating file system instances.
 *
 * The caller is responsible for releasing its own hold on the mount point
 * vp (this routine does its own hold when necessary).
 * Also note that for remounts, the mount point vp should be the vnode for
 * the root of the file system rather than the vnode that the file system
 * is mounted on top of.
 */
int
domount(char *fsname, struct mounta *uap, vnode_t *vp, struct cred *credp,
        struct vfs **vfspp)
{
        struct vfssw    *vswp;
        vfsops_t        *vfsops;
        struct vfs      *vfsp;
        struct vnode    *bvp;
        dev_t           bdev = 0;
        mntopts_t       mnt_mntopts;
        int             error = 0;
        int             copyout_error = 0;
        int             ovflags;
        char            *opts = uap->optptr;
        char            *inargs = opts;
        int             optlen = uap->optlen;
        int             remount;
        int             rdonly;
        int             nbmand = 0;
        int             delmip = 0;
        int             addmip = 0;
        int             splice = ((uap->flags & MS_NOSPLICE) == 0);
        int             fromspace = (uap->flags & MS_SYSSPACE) ?
            UIO_SYSSPACE : UIO_USERSPACE;
        char            *resource = NULL, *mountpt = NULL;
        refstr_t        *oldresource, *oldmntpt;
        struct pathname pn, rpn;
        vsk_anchor_t    *vskap;
        char fstname[FSTYPSZ];
        zone_t          *zone;

        /*
         * The v_flag value for the mount point vp is permanently set
         * to VVFSLOCK so that no one bypasses the vn_vfs*locks routine
         * for mount point locking.
         */
        mutex_enter(&vp->v_lock);
        vp->v_flag |= VVFSLOCK;
        mutex_exit(&vp->v_lock);

        mnt_mntopts.mo_count = 0;
        /*
         * Find the ops vector to use to invoke the file system-specific mount
         * method.  If the fsname argument is non-NULL, use it directly.
         * Otherwise, dig the file system type information out of the mount
         * arguments.
         *
         * A side effect is to hold the vfssw entry.
         *
         * Mount arguments can be specified in several ways, which are
         * distinguished by flag bit settings.  The preferred way is to set
         * MS_OPTIONSTR, indicating an 8 argument mount with the file system
         * type supplied as a character string and the last two arguments
         * being a pointer to a character buffer and the size of the buffer.
         * On entry, the buffer holds a null terminated list of options; on
         * return, the string is the list of options the file system
         * recognized. If MS_DATA is set arguments five and six point to a
         * block of binary data which the file system interprets.
         * A further wrinkle is that some callers don’t set MS_FSS and MS_DATA
         * consistently with these conventions.  To handle them, we check to
         * see whether the pointer to the file system name has a numeric value
         * less than 256.  If so, we treat it as an index.
         */
        if (fsname != NULL) {
                if ((vswp = vfs_getvfssw(fsname)) == NULL) {
                        return (EINVAL);
                }
        } else if (uap->flags & (MS_OPTIONSTR | MS_DATA | MS_FSS)) {
                size_t n;
                uint_t fstype;

                fsname = fstname;

                if ((fstype = (uintptr_t)uap->fstype) < 256) {
                        RLOCK_VFSSW();
                        if (fstype == 0 || fstype >= nfstype ||
                            !ALLOCATED_VFSSW(&vfssw[fstype])) {
                                RUNLOCK_VFSSW();
                                return (EINVAL);
                        }
                        (void) strcpy(fsname, vfssw[fstype].vsw_name);
                        RUNLOCK_VFSSW();
                        if ((vswp = vfs_getvfssw(fsname)) == NULL)
                                return (EINVAL);
                } else {
                        /*
```

```
                         * Handle either kernel or user address space.
                         */
                        if (uap->flags & MS_SYSSPACE) {
                                error = copystr(uap->fstype, fsname,
                                    FSTYPSZ, &n);
                        } else {
                                error = copyinstr(uap->fstype, fsname,
                                    FSTYPSZ, &n);
                        }
                        if (error) {
                                if (error == ENAMETOOLONG)
                                        return (EINVAL);
                                return (error);
                        }
                        if ((vswp = vfs_getvfssw(fsname)) == NULL)
                                return (EINVAL);
                }
        } else {
                if ((vswp = vfs_getvfsswbyvfsops(vfs_getops(rootvfs))) == NULL)
                        return (EINVAL);
                fsname = vswp->vsw_name;
        }
        if (!VFS_INSTALLED(vswp))
                return (EINVAL);

        if ((error = secpolicy_fs_allowed_mount(fsname)) != 0)  {
                vfs_unrefvfssw(vswp);
                return (error);
        }

        vfsops = &vswp->vsw_vfsops;

        vfs_copyopttbl(&vswp->vsw_optproto, &mnt_mntopts);
        /*
         * Fetch mount options and parse them for generic vfs options
         */
        if (uap->flags & MS_OPTIONSTR) {
                /*
                 * Limit the buffer size
                 */
                if (optlen < 0 || optlen > MAX_MNTOPT_STR) {
                        error = EINVAL;
                        goto errout;
                }
                if ((uap->flags & MS_SYSSPACE) == 0) {
                        inargs = kmem_alloc(MAX_MNTOPT_STR, KM_SLEEP);
                        inargs[0] = ’\0’;
                        if (optlen) {
                                error = copyinstr(opts, inargs, (size_t)optlen,
                                    NULL);
                                if (error) {
                                        goto errout;
                                }
                        }
                }
                vfs_parsemntopts(&mnt_mntopts, inargs, 0);
        }
        /*
         * Flag bits override the options string.
         */
        if (uap->flags & MS_REMOUNT)
                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_REMOUNT, NULL, 0, 0);
        if (uap->flags & MS_RDONLY)
                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_RO, NULL, 0, 0);
        if (uap->flags & MS_NOSUID)
                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL, 0, 0);
        /*
         * Check if this is a remount; must be set in the option string and
         * the file system must support a remount option.
         */
        if (remount = vfs_optionisset_nolock(&mnt_mntopts,
            MNTOPT_REMOUNT, NULL)) {
                if (!(vswp->vsw_flag & VSW_CANREMOUNT)) {
                        error = ENOTSUP;
                        goto errout;
                }
                uap->flags |= MS_REMOUNT;
        }

        /*
         * uap->flags and vfs_optionisset() should agree.
         */
        if (rdonly = vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_RO, NULL)) {
                uap->flags |= MS_RDONLY;
        }
        if (vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL)) {
                uap->flags |= MS_NOSUID;
        }
        nbmand = vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_NBMAND, NULL);
        ASSERT(splice || !remount);
        /*
         * If we are splicing the fs into the namespace,
         * perform mount point checks.
         *
         * We want to resolve the path for the mount point to eliminate
         * '.' and ".." and symlinks in mount points; we can't do the
         * same for the resource string, since it would turn
         * "/dev/dsk/c0t0d0s0" into "/devices/pci@...".  We need to do
         * this before grabbing vn_vfswlock(), because otherwise we
         * would deadlock with lookuppn().
         */
        if (splice) {
                ASSERT(vp->v_count > 0);

                /*
                 * Pick up mount point and device from appropriate space.
                 */
                if (pn_get(uap->spec, fromspace, &pn) == 0) {
                        resource = kmem_alloc(pn.pn_pathlen + 1,
                            KM_SLEEP);
                        (void) strcpy(resource, pn.pn_path);
                        pn_free(&pn);
                }
                /*
                 * Do a lookupname prior to taking the
                 * writelock. Mark this as completed if
                 * successful for later cleanup and addition to
                 * the mount in progress table.
                 */
                if ((uap->flags & MS_GLOBAL) == 0 &&
                    lookupname(uap->spec, fromspace,
                    FOLLOW, NULL, &bvp) == 0) {
                        addmip = 1;
                }

                if ((error = pn_get(uap->dir, fromspace, &pn)) == 0) {
                        pathname_t *pnp;

                        if (*pn.pn_path != '/') {
                                error = EINVAL;
                                pn_free(&pn);
```

```
                                goto errout;
                        }
                        pn_alloc(&rpn);
                        /*
                         * Kludge to prevent autofs from deadlocking with
                         * itself when it calls domount().
                         *
                         * If autofs is calling, it is because it is doing
                         * (autofs) mounts in the process of an NFS mount.  A
                         * lookuppn() here would cause us to block waiting for
                         * said NFS mount to complete, which can't since this
                         * is the thread that was supposed to doing it.
                         */
                        if (fromspace == UIO_USERSPACE) {
                                if ((error = lookuppn(&pn, &rpn, FOLLOW, NULL,
                                    NULL)) == 0) {
                                        pnp = &rpn;
                                } else {
                                        /*
                                         * The file disappeared or otherwise
                                         * became inaccessible since we opened
                                         * it; might as well fail the mount
                                         * since the mount point is no longer
                                         * accessible.
                                         */
                                        pn_free(&rpn);
                                        pn_free(&pn);
                                        goto errout;
                                }
                        } else {
                                pnp = &pn;
                        }
                        mountpt = kmem_alloc(pnp->pn_pathlen + 1, KM_SLEEP);
                        (void) strcpy(mountpt, pnp->pn_path);

                        /*
                         * If the addition of the zone's rootpath
                         * would push us over a total path length
                         * of MAXPATHLEN, we fail the mount with
                         * ENAMETOOLONG, which is what we would have
                         * gotten if we were trying to perform the same
                         * mount in the global zone.
                         *
                         * strlen() doesn't count the trailing
                         * '\0', but zone_rootpathlen counts both a
                         * trailing '/' and the terminating '\0'.
                         */
                        if ((curproc->p_zone->zone_rootpathlen - 1 +
                            strlen(mountpt)) > MAXPATHLEN ||
                            (resource != NULL &&
                            (curproc->p_zone->zone_rootpathlen - 1 +
                            strlen(resource)) > MAXPATHLEN)) {
                                error = ENAMETOOLONG;
                        }

                        pn_free(&rpn);
                        pn_free(&pn);
                }

                if (error)
                        goto errout;

                /*
                 * Prevent path name resolution from proceeding past
                 * the mount point.
                 */
                if (vn_vfswlock(vp) != 0) {
                        error = EBUSY;
                        goto errout;
                }

                /*
                 * Verify that it’s legitimate to establish a mount on
                 * the prospective mount point.
                 */
                if (vn_mountedvfs(vp) != NULL) {
                        /*
                         * The mount point lock was obtained after some
                         * other thread raced through and established a mount.
                         */
                        vn_vfsunlock(vp);
                        error = EBUSY;
                        goto errout;
                }
                if (vp->v_flag & VNOMOUNT) {
                        vn_vfsunlock(vp);
                        error = EINVAL;
                        goto errout;
                }
        }
        if ((uap->flags & (MS_DATA | MS_OPTIONSTR)) == 0) {
                uap->dataptr = NULL;
                uap->datalen = 0;
        }

        /*
         * If this is a remount, we don’t want to create a new VFS.
         * Instead, we pass the existing one with a remount flag.
         */
        if (remount) {
                /*
                 * Confirm that the mount point is the root vnode of the
                 * file system that is being remounted.
                 * This can happen if the user specifies a different
                 * mount point directory pathname in the (re)mount command.
                 *
                 * Code below can only be reached if splice is true, so it’s
                 * safe to do vn_vfsunlock() here.
                 */
                if ((vp->v_flag & VROOT) == 0) {
                        vn_vfsunlock(vp);
                        error = ENOENT;
                        goto errout;
                }
                /*
                 * Disallow making file systems read-only unless file system
                 * explicitly allows it in its vfssw.  Ignore other flags.
                 */
                if (rdonly && vn_is_readonly(vp) == 0 &&
                    (vswp->vsw_flag & VSW_CANRWRO) == 0) {
                        vn_vfsunlock(vp);
                        error = EINVAL;
                        goto errout;
                }
                /*
                 * Disallow changing the NBMAND disposition of the file
                 * system on remounts.
                 */
                if ((nbmand && ((vp->v_vfsp->vfs_flag & VFS_NBMAND) == 0)) ||
                    (!nbmand && (vp->v_vfsp->vfs_flag & VFS_NBMAND))) {
                        vn_vfsunlock(vp);
                        error = EINVAL;
```

```
                        goto errout;
                }
                vfsp = vp->v_vfsp;
                ovflags = vfsp->vfs_flag;
                vfsp->vfs_flag |= VFS_REMOUNT;
                vfsp->vfs_flag &= ~VFS_RDONLY;
        } else {
                vfsp = vfs_alloc(KM_SLEEP);
                VFS_INIT(vfsp, vfsops, NULL);
        }

        VFS_HOLD(vfsp);

        if ((error = lofi_add(fsname, vfsp, &mnt_mntopts, uap)) != 0) {
                if (!remount) {
                        if (splice)
                                vn_vfsunlock(vp);
                        vfs_free(vfsp);
                } else {
                        vn_vfsunlock(vp);
                        VFS_RELE(vfsp);
                }
                goto errout;
        }

        /*
         * PRIV_SYS_MOUNT doesn’t mean you can become root.
         */
        if (vfsp->vfs_lofi_minor != 0) {
                uap->flags |= MS_NOSUID;
                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL, 0, 0);
        }

        /*
         * The vfs_reflock is not used anymore the code below explicitly
         * holds it preventing others accesing it directly.
         */
        if ((sema_tryp(&vfsp->vfs_reflock) == 0) &&
            !(vfsp->vfs_flag & VFS_REMOUNT))
                cmn_err(CE_WARN,
                    "mount type %s couldn’t get vfs_reflock", vswp->vsw_name);

        /*
         * Lock the vfs. If this is a remount we want to avoid spurious umount
         * failures that happen as a side-effect of fsflush() and other mount
         * and unmount operations that might be going on simultaneously and
         * may have locked the vfs currently. To not return EBUSY immediately
         * here we use vfs_lock_wait() instead vfs_lock() for the remount case.
         */
        if (!remount) {
                if (error = vfs_lock(vfsp)) {
                        vfsp->vfs_flag = ovflags;

                        lofi_remove(vfsp);

                        if (splice)
                                vn_vfsunlock(vp);
                        vfs_free(vfsp);
                        goto errout;
                }
        } else {
                vfs_lock_wait(vfsp);
        }

        /*
         * Add device to mount in progress table, global mounts require special
         * handling. It is possible that we have already done the lookupname
         * on a spliced, non-global fs. If so, we don't want to do it again
         * since we cannot do a lookupname after taking the
         * wlock above. This case is for a non-spliced, non-global filesystem.
         */
        if (!addmip) {
                if ((uap->flags & MS_GLOBAL) == 0 &&
                    lookupname(uap->spec, fromspace, FOLLOW, NULL, &bvp) == 0) {
                        addmip = 1;
                }
        }

        if (addmip) {
                vnode_t *lvp = NULL;

                error = vfs_get_lofi(vfsp, &lvp);
                if (error > 0) {
                        lofi_remove(vfsp);

                        if (splice)
                                vn_vfsunlock(vp);
                        vfs_unlock(vfsp);

                        if (remount) {
                                VFS_RELE(vfsp);
                        } else {
                                vfs_free(vfsp);
                        }

                        goto errout;
                } else if (error == -1) {
                        bdev = bvp->v_rdev;
                        VN_RELE(bvp);
                } else {
                        bdev = lvp->v_rdev;
                        VN_RELE(lvp);
                        VN_RELE(bvp);
                }

                vfs_addmip(bdev, vfsp);
                addmip = 0;
                delmip = 1;
        }
        /*
         * Invalidate cached entry for the mount point.
         */
        if (splice)
                dnlc_purge_vp(vp);

        /*
         * If have an option string but the filesystem doesn't supply a
         * prototype options table, create a table with the global
         * options and sufficient room to accept all the options in the
         * string.  Then parse the passed in option string
         * accepting all the options in the string.  This gives us an
         * option table with all the proper cancel properties for the
         * global options.
         *
         * Filesystems that supply a prototype options table are handled
         * earlier in this function.
         */
        if (uap->flags & MS_OPTIONSTR) {
                if (!(vswp->vsw_flag & VSW_HASPROTO)) {
                        mntopts_t tmp_mntopts;

                        tmp_mntopts.mo_count = 0;
```

```
                        vfs_createopttbl_extend(&tmp_mntopts, inargs,
                            &mnt_mntopts);
                        vfs_parsemntopts(&tmp_mntopts, inargs, 1);
                        vfs_swapopttbl_nolock(&mnt_mntopts, &tmp_mntopts);
                        vfs_freeopttbl(&tmp_mntopts);
                }
        }

        /*
         * Serialize with zone state transitions.
         * See vfs_list_add; zone mounted into is:
         *      zone_find_by_path(refstr_value(vfsp->vfs_mntpt))
         * not the zone doing the mount (curproc->p_zone), but if we're already
         * inside a NGZ, then we know what zone we are.
         */
        if (INGLOBALZONE(curproc)) {
                zone = zone_find_by_path(mountpt);
                ASSERT(zone != NULL);
        } else {
                zone = curproc->p_zone;
                /*
                 * zone_find_by_path does a hold, so do one here too so that
                 * we can do a zone_rele after mount_completed.
                 */
                zone_hold(zone);
        }
        mount_in_progress(zone);
        /*
         * Instantiate (or reinstantiate) the file system.  If appropriate,
         * splice it into the file system name space.
         *
         * We want VFS_MOUNT() to be able to override the vfs_resource
         * string if necessary (ie, mntfs), and also for a remount to
         * change the same (necessary when remounting '/' during boot).
         * So we set up vfs_mntpt and vfs_resource to what we think they
         * should be, then hand off control to VFS_MOUNT() which can
         * override this.
         *
         * For safety's sake, when changing vfs_resource or vfs_mntpt of
         * a vfs which is on the vfs list (i.e. during a remount), we must
         * never set those fields to NULL. Several bits of code make
         * assumptions that the fields are always valid.
         */
        vfs_swapopttbl(&mnt_mntopts, &vfsp->vfs_mntopts);
        if (remount) {
                if ((oldresource = vfsp->vfs_resource) != NULL)
                        refstr_hold(oldresource);
                if ((oldmntpt = vfsp->vfs_mntpt) != NULL)
                        refstr_hold(oldmntpt);
        }
        vfs_setresource(vfsp, resource, 0);
        vfs_setmntpoint(vfsp, mountpt, 0);

        /*
         * going to mount on this vnode, so notify.
         */
        vnevent_mountedover(vp, NULL);
        error = VFS_MOUNT(vfsp, vp, uap, credp);

        if (uap->flags & MS_RDONLY)
                vfs_setmntopt(vfsp, MNTOPT_RO, NULL, 0);
        if (uap->flags & MS_NOSUID)
                vfs_setmntopt(vfsp, MNTOPT_NOSUID, NULL, 0);
        if (uap->flags & MS_GLOBAL)
                vfs_setmntopt(vfsp, MNTOPT_GLOBAL, NULL, 0);
        if (error) {
                lofi_remove(vfsp);

                if (remount) {
                        /* put back pre-remount options */
                        vfs_swapopttbl(&mnt_mntopts, &vfsp->vfs_mntopts);
                        vfs_setmntpoint(vfsp, refstr_value(oldmntpt),
                            VFSSP_VERBATIM);
                        if (oldmntpt)
                                refstr_rele(oldmntpt);
                        vfs_setresource(vfsp, refstr_value(oldresource),
                            VFSSP_VERBATIM);
                        if (oldresource)
                                refstr_rele(oldresource);
                        vfsp->vfs_flag = ovflags;
                        vfs_unlock(vfsp);
                        VFS_RELE(vfsp);
                } else {
                        vfs_unlock(vfsp);
                        vfs_freemnttab(vfsp);
                        vfs_free(vfsp);
                }
        } else {
                /*
                 * Set the mount time to now
                 */
                vfsp->vfs_mtime = ddi_get_time();
                if (remount) {
                        vfsp->vfs_flag &= ~VFS_REMOUNT;
                        if (oldresource)
                                refstr_rele(oldresource);
                        if (oldmntpt)
                                refstr_rele(oldmntpt);
                } else if (splice) {
                        /*
                         * Link vfsp into the name space at the mount
                         * point. Vfs_add() is responsible for
                         * holding the mount point which will be
                         * released when vfs_remove() is called.
                         */
                        vfs_add(vp, vfsp, uap->flags);
                } else {
                        /*
                         * Hold the reference to file system which is
                         * not linked into the name space.
                         */
                        vfsp->vfs_zone = NULL;
                        VFS_HOLD(vfsp);
                        vfsp->vfs_vnodecovered = NULL;
                }
                /*
                 * Set flags for global options encountered
                 */
                if (vfs_optionisset(vfsp, MNTOPT_RO, NULL))
                        vfsp->vfs_flag |= VFS_RDONLY;
                else
                        vfsp->vfs_flag &= ~VFS_RDONLY;
                if (vfs_optionisset(vfsp, MNTOPT_NOSUID, NULL)) {
                        vfsp->vfs_flag |= (VFS_NOSETUID|VFS_NODEVICES);
                } else {
                        if (vfs_optionisset(vfsp, MNTOPT_NODEVICES, NULL))
                                vfsp->vfs_flag |= VFS_NODEVICES;
                        else
                                vfsp->vfs_flag &= ~VFS_NODEVICES;
                        if (vfs_optionisset(vfsp, MNTOPT_NOSETUID, NULL))
                                vfsp->vfs_flag |= VFS_NOSETUID;
```

```
                        else
                                vfsp->vfs_flag &= ~VFS_NOSETUID;
                }
                if (vfs_optionisset(vfsp, MNTOPT_NBMAND, NULL))
                        vfsp->vfs_flag |= VFS_NBMAND;
                else
                        vfsp->vfs_flag &= ~VFS_NBMAND;

                if (vfs_optionisset(vfsp, MNTOPT_XATTR, NULL))
                        vfsp->vfs_flag |= VFS_XATTR;
                else
                        vfsp->vfs_flag &= ~VFS_XATTR;

                if (vfs_optionisset(vfsp, MNTOPT_NOEXEC, NULL))
                        vfsp->vfs_flag |= VFS_NOEXEC;
                else
                        vfsp->vfs_flag &= ~VFS_NOEXEC;

                /*
                 * Now construct the output option string of options
                 * we recognized.
                 */
                if (uap->flags & MS_OPTIONSTR) {
                        vfs_list_read_lock();
                        copyout_error = vfs_buildoptionstr(
                            &vfsp->vfs_mntopts, inargs, optlen);
                        vfs_list_unlock();
                        if (copyout_error == 0 &&
                            (uap->flags & MS_SYSSPACE) == 0) {
                                copyout_error = copyoutstr(inargs, opts,
                                    optlen, NULL);
                        }
                }

                /*
                 * If this isn’t a remount, set up the vopstats before
                 * anyone can touch this. We only allow spliced file
                 * systems (file systems which are in the namespace) to
                 * have the VFS_STATS flag set.
                 * NOTE: PxFS mounts the underlying file system with
                 * MS_NOSPLICE set and copies those vfs_flags to its private
                 * vfs structure. As a result, PxFS should never have
                 * the VFS_STATS flag or else we might access the vfs
                 * statistics-related fields prior to them being
                 * properly initialized.
                 */
                if (!remount && (vswp->vsw_flag & VSW_STATS) && splice) {
                        initialize_vopstats(&vfsp->vfs_vopstats);
                        /*
                         * We need to set vfs_vskap to NULL because there’s
                         * a chance it won’t be set below.  This is checked
                         * in teardown_vopstats() so we can’t have garbage.
                         */
                        vfsp->vfs_vskap = NULL;
                        vfsp->vfs_flag |= VFS_STATS;
                        vfsp->vfs_fstypevsp = get_fstype_vopstats(vfsp, vswp);
                }

                if (vswp->vsw_flag & VSW_XID)
                        vfsp->vfs_flag |= VFS_XID;

                vfs_unlock(vfsp);
        }
        mount_completed(zone);
        zone_rele(zone);
        if (splice)
```

```
                vn_vfsunlock(vp);

        if ((error == 0) && (copyout_error == 0)) {
                if (!remount) {
                        /*
                         * Don’t call get_vskstat_anchor() while holding
                         * locks since it allocates memory and calls
                         * VFS_STATVFS().  For NFS, the latter can generate
                         * an over-the-wire call.
                         */
                        vskap = get_vskstat_anchor(vfsp);
                        /* Only take the lock if we have something to do */
                        if (vskap != NULL) {
                                vfs_lock_wait(vfsp);
                                if (vfsp->vfs_flag & VFS_STATS) {
                                        vfsp->vfs_vskap = vskap;
                                }
                                vfs_unlock(vfsp);
                        }
                }
                /* Return vfsp to caller. */
                *vfspp = vfsp;
        }
errout:
        vfs_freeopttbl(&mnt_mntopts);
        if (resource != NULL)
                kmem_free(resource, strlen(resource) + 1);
        if (mountpt != NULL)
                kmem_free(mountpt, strlen(mountpt) + 1);
        /*
         * It is possible we errored prior to adding to mount in progress
         * table. Must free vnode we acquired with successful lookupname.
         */
        if (addmip)
                VN_RELE(bvp);
        if (delmip)
                vfs_delmip(vfsp);
        ASSERT(vswp != NULL);
        vfs_unrefvfssw(vswp);
        if (inargs != opts)
                kmem_free(inargs, MAX_MNTOPT_STR);
        if (copyout_error) {
                lofi_remove(vfsp);
                VFS_RELE(vfsp);
                error = copyout_error;
        }
        return (error);
}

static void
vfs_setpath(
    struct vfs *vfsp,           /* vfs being updated */
    refstr_t **refp,            /* Ref-count string to contain the new path */
    const char *newpath,        /* Path to add to refp (above) */
    uint32_t flag)              /* flag */
{
        size_t len;
        refstr_t *ref;
        zone_t *zone = curproc->p_zone;
        char *sp;
        int have_list_lock = 0;

        ASSERT(!VFS_ON_LIST(vfsp) || vfs_lock_held(vfsp));

        /*
         * New path must be less than MAXPATHLEN because mntfs
```

```
         * will only display up to MAXPATHLEN bytes. This is currently
         * safe, because domount() uses pn_get(), and other callers
         * similarly cap the size to fewer than MAXPATHLEN bytes.
         */

        ASSERT(strlen(newpath) < MAXPATHLEN);

        /* mntfs requires consistency while vfs list lock is held */

        if (VFS_ON_LIST(vfsp)) {
                have_list_lock = 1;
                vfs_list_lock();
        }

        if (*refp != NULL)
                refstr_rele(*refp);

        /*
         * If we are in a non-global zone then we prefix the supplied path,
         * newpath, with the zone’s root path, with two exceptions. The first
         * is where we have been explicitly directed to avoid doing so; this
         * will be the case following a failed remount, where the path supplied
         * will be a saved version which must now be restored. The second
         * exception is where newpath is not a pathname but a descriptive name,
         * e.g. "procfs".
         */
        if (zone == global_zone || (flag & VFSSP_VERBATIM) || *newpath != ’/’) {
                ref = refstr_alloc(newpath);
                goto out;
        }

        /*
         * Truncate the trailing ’/’ in the zoneroot, and merge
         * in the zone’s rootpath with the "newpath" (resource
         * or mountpoint) passed in.
         *
         * The size of the required buffer is thus the size of
         * the buffer required for the passed-in newpath
         * (strlen(newpath) + 1), plus the size of the buffer
         * required to hold zone_rootpath (zone_rootpathlen)
         * minus one for one of the now-superfluous NUL
         * terminations, minus one for the trailing ’/’.
         *
         * That gives us:
         *
         * (strlen(newpath) + 1) + zone_rootpathlen - 1 - 1
         *
         * Which is what we have below.
         */

        len = strlen(newpath) + zone->zone_rootpathlen - 1;
        sp = kmem_alloc(len, KM_SLEEP);

        /*
         * Copy everything including the trailing slash, which
         * we then overwrite with the NUL character.
         */

        (void) strcpy(sp, zone->zone_rootpath);
        sp[zone->zone_rootpathlen - 2] = ’\0’;
        (void) strcat(sp, newpath);

        ref = refstr_alloc(sp);
        kmem_free(sp, len);
out:
        *refp = ref;
```

```
	if (have_list_lock) {
		vfs_mnttab_modtimeupd();
		vfs_list_unlock();
	}
}

/*
 * Record a mounted resource name in a vfs structure.
 * If vfsp is already mounted, caller must hold the vfs lock.
 */
void
vfs_setresource(struct vfs *vfsp, const char *resource, uint32_t flag)
{
	if (resource == NULL || resource[0] == ’\0’)
		resource = VFS_NORESOURCE;
	vfs_setpath(vfsp, &vfsp->vfs_resource, resource, flag);
}

/*
 * Record a mount point name in a vfs structure.
 * If vfsp is already mounted, caller must hold the vfs lock.
 */
void
vfs_setmntpoint(struct vfs *vfsp, const char *mntpt, uint32_t flag)
{
	if (mntpt == NULL || mntpt[0] == ’\0’)
		mntpt = VFS_NOMNTPT;
	vfs_setpath(vfsp, &vfsp->vfs_mntpt, mntpt, flag);
}

/* Returns the vfs_resource. Caller must call refstr_rele() when finished. */

refstr_t *
vfs_getresource(const struct vfs *vfsp)
{
	refstr_t *resource;

	vfs_list_read_lock();
	resource = vfsp->vfs_resource;
	refstr_hold(resource);
	vfs_list_unlock();

	return (resource);
}

/* Returns the vfs_mntpt. Caller must call refstr_rele() when finished. */

refstr_t *
vfs_getmntpoint(const struct vfs *vfsp)
{
	refstr_t *mntpt;

	vfs_list_read_lock();
	mntpt = vfsp->vfs_mntpt;
	refstr_hold(mntpt);
	vfs_list_unlock();

	return (mntpt);
}

/*
 * Create an empty options table with enough empty slots to hold all
 * The options in the options string passed as an argument.
 * Potentially prepend another options table.
 *
```

```
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect mops.
 */
static void
vfs_createopttbl_extend(mntopts_t *mops, const char *opts,
    const mntopts_t *mtmpl)
{
	const char *s = opts;
	uint_t count;

	if (opts == NULL || *opts == ’\0’) {
		count = 0;
	} else {
		count = 1;

		/*
		 * Count number of options in the string
		 */
		for (s = strchr(s, ’,’); s != NULL; s = strchr(s, ’,’)) {
			count++;
			s++;
		}
	}
	vfs_copyopttbl_extend(mtmpl, mops, count);
}

/*
 * Create an empty options table with enough empty slots to hold all
 * The options in the options string passed as an argument.
 *
 * This function is *not* for general use by filesystems.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect mops.
 */
void
vfs_createopttbl(mntopts_t *mops, const char *opts)
{
	vfs_createopttbl_extend(mops, opts, NULL);
}


/*
 * Swap two mount options tables
 */
static void
vfs_swapopttbl_nolock(mntopts_t *optbl1, mntopts_t *optbl2)
{
	uint_t tmpcnt;
	mntopt_t *tmplist;

	tmpcnt = optbl2->mo_count;
	tmplist = optbl2->mo_list;
	optbl2->mo_count = optbl1->mo_count;
	optbl2->mo_list = optbl1->mo_list;
	optbl1->mo_count = tmpcnt;
	optbl1->mo_list = tmplist;
}

static void
vfs_swapopttbl(mntopts_t *optbl1, mntopts_t *optbl2)
{
	vfs_list_lock();
	vfs_swapopttbl_nolock(optbl1, optbl2);
	vfs_mnttab_modtimeupd();
	vfs_list_unlock();
```

```
}

static char **
vfs_copycancelopt_extend(char **const moc, int extend)
{
        int i = 0;
        int j;
        char **result;

        if (moc != NULL) {
                for (; moc[i] != NULL; i++)
                        /* count number of options to cancel */;
        }

        if (i + extend == 0)
                return (NULL);

        result = kmem_alloc((i + extend + 1) * sizeof (char *), KM_SLEEP);

        for (j = 0; j < i; j++) {
                result[j] = kmem_alloc(strlen(moc[j]) + 1, KM_SLEEP);
                (void) strcpy(result[j], moc[j]);
        }
        for (; j <= i + extend; j++)
                result[j] = NULL;

        return (result);
}

static void
vfs_copyopt(const mntopt_t *s, mntopt_t *d)
{
        char *sp, *dp;

        d->mo_flags = s->mo_flags;
        d->mo_data = s->mo_data;
        sp = s->mo_name;
        if (sp != NULL) {
                dp = kmem_alloc(strlen(sp) + 1, KM_SLEEP);
                (void) strcpy(dp, sp);
                d->mo_name = dp;
        } else {
                d->mo_name = NULL; /* should never happen */
        }

        d->mo_cancel = vfs_copycancelopt_extend(s->mo_cancel, 0);

        sp = s->mo_arg;
        if (sp != NULL) {
                dp = kmem_alloc(strlen(sp) + 1, KM_SLEEP);
                (void) strcpy(dp, sp);
                d->mo_arg = dp;
        } else {
                d->mo_arg = NULL;
        }
}

/*
 * Copy a mount options table, possibly allocating some spare
 * slots at the end.  It is permissible to copy_extend the NULL table.
 */
static void
vfs_copyopttbl_extend(const mntopts_t *smo, mntopts_t *dmo, int extra)
{
        uint_t i, count;
        mntopt_t *motbl;
```

```

        /*
         * Clear out any existing stuff in the options table being initialized
         */
        vfs_freeopttbl(dmo);
        count = (smo == NULL) ? 0 : smo->mo_count;
        if ((count + extra) == 0)       /* nothing to do */
                return;
        dmo->mo_count = count + extra;
        motbl = kmem_zalloc((count + extra) * sizeof (mntopt_t), KM_SLEEP);
        dmo->mo_list = motbl;
        for (i = 0; i < count; i++) {
                vfs_copyopt(&smo->mo_list[i], &motbl[i]);
        }
        for (i = count; i < count + extra; i++) {
                motbl[i].mo_flags = MO_EMPTY;
        }
}

/*
 * Copy a mount options table.
 *
 * This function is *not* for general use by filesystems.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect smo and dmo.
 */
void
vfs_copyopttbl(const mntopts_t *smo, mntopts_t *dmo)
{
        vfs_copyopttbl_extend(smo, dmo, 0);
}

static char **
vfs_mergecancelopts(const mntopt_t *mop1, const mntopt_t *mop2)
{
        int c1 = 0;
        int c2 = 0;
        char **result;
        char **sp1, **sp2, **dp;

        /*
         * First we count both lists of cancel options.
         * If either is NULL or has no elements, we return a copy of
         * the other.
         */
        if (mop1->mo_cancel != NULL) {
                for (; mop1->mo_cancel[c1] != NULL; c1++)
                        /* count cancel options in mop1 */;
        }

        if (c1 == 0)
                return (vfs_copycancelopt_extend(mop2->mo_cancel, 0));

        if (mop2->mo_cancel != NULL) {
                for (; mop2->mo_cancel[c2] != NULL; c2++)
                        /* count cancel options in mop2 */;
        }

        result = vfs_copycancelopt_extend(mop1->mo_cancel, c2);

        if (c2 == 0)
                return (result);

        /*
         * When we get here, we’ve got two sets of cancel options;
         * we need to merge the two sets.  We know that the result
         * array has "c1+c2+1" entries and in the end we might shrink
         * it.
         * Result now has a copy of the c1 entries from mop1; we’ll
         * now lookup all the entries of mop2 in mop1 and copy it if
         * it is unique.
         * This operation is O(n^2) but it’s only called once per
         * filesystem per duplicate option.  This is a situation
         * which doesn’t arise with the filesystems in ON and
         * n is generally 1.
         */

        dp = &result[c1];
        for (sp2 = mop2->mo_cancel; *sp2 != NULL; sp2++) {
                for (sp1 = mop1->mo_cancel; *sp1 != NULL; sp1++) {
                        if (strcmp(*sp1, *sp2) == 0)
                                break;
                }
                if (*sp1 == NULL) {
                        /*
                         * Option *sp2 not found in mop1, so copy it.
                         * The calls to vfs_copycancelopt_extend()
                         * guarantee that there’s enough room.
                         */
                        *dp = kmem_alloc(strlen(*sp2) + 1, KM_SLEEP);
                        (void) strcpy(*dp++, *sp2);
                }
        }
        if (dp != &result[c1+c2]) {
                size_t bytes = (dp - result + 1) * sizeof (char *);
                char **nres = kmem_alloc(bytes, KM_SLEEP);

                bcopy(result, nres, bytes);
                kmem_free(result, (c1 + c2 + 1) * sizeof (char *));
                result = nres;
        }
        return (result);
}

/*
 * Merge two mount option tables (outer and inner) into one.  This is very
 * similar to "merging" global variables and automatic variables in C.
 *
 * This isn’t (and doesn’t have to be) fast.
 *
 * This function is *not* for general use by filesystems.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect omo, imo & dmo.
 */
void
vfs_mergeopttbl(const mntopts_t *omo, const mntopts_t *imo, mntopts_t *dmo)
{
        uint_t i, count;
        mntopt_t *mop, *motbl;
        uint_t freeidx;

        /*
         * First determine how much space we need to allocate.
         */
        count = omo->mo_count;
        for (i = 0; i < imo->mo_count; i++) {
                if (imo->mo_list[i].mo_flags & MO_EMPTY)
                        continue;
                if (vfs_hasopt(omo, imo->mo_list[i].mo_name) == NULL)
                        count++;
```

```
        }
        ASSERT(count >= omo->mo_count &&
            count <= omo->mo_count + imo->mo_count);
        motbl = kmem_alloc(count * sizeof (mntopt_t), KM_SLEEP);
        for (i = 0; i < omo->mo_count; i++)
                vfs_copyopt(&omo->mo_list[i], &motbl[i]);
        freeidx = omo->mo_count;
        for (i = 0; i < imo->mo_count; i++) {
                if (imo->mo_list[i].mo_flags & MO_EMPTY)
                        continue;
                if ((mop = vfs_hasopt(omo, imo->mo_list[i].mo_name)) != NULL) {
                        char **newcanp;
                        uint_t index = mop - omo->mo_list;

                        newcanp = vfs_mergecancelopts(mop, &motbl[index]);

                        vfs_freeopt(&motbl[index]);
                        vfs_copyopt(&imo->mo_list[i], &motbl[index]);

                        vfs_freecancelopt(motbl[index].mo_cancel);
                        motbl[index].mo_cancel = newcanp;
                } else {
                        /*
                         * If it’s a new option, just copy it over to the first
                         * free location.
                         */
                        vfs_copyopt(&imo->mo_list[i], &motbl[freeidx++]);
                }
        }
        dmo->mo_count = count;
        dmo->mo_list = motbl;
}

/*
 * Functions to set and clear mount options in a mount options table.
 */

/*
 * Clear a mount option, if it exists.
 *
 * The update_mnttab arg indicates whether mops is part of a vfs that is on
 * the vfs list.
 */
static void
vfs_clearmntopt_nolock(mntopts_t *mops, const char *opt, int update_mnttab)
{
        struct mntopt *mop;
        uint_t i, count;

        ASSERT(!update_mnttab || RW_WRITE_HELD(&vfslist));

        count = mops->mo_count;
        for (i = 0; i < count; i++) {
                mop = &mops->mo_list[i];

                if (mop->mo_flags & MO_EMPTY)
                        continue;
                if (strcmp(opt, mop->mo_name))
                        continue;
                mop->mo_flags &= ~MO_SET;
                if (mop->mo_arg != NULL) {
                        kmem_free(mop->mo_arg, strlen(mop->mo_arg) + 1);
                }
                mop->mo_arg = NULL;
                if (update_mnttab)
                        vfs_mnttab_modtimeupd();
                break;
        }
}

void
vfs_clearmntopt(struct vfs *vfsp, const char *opt)
{
        int gotlock = 0;

        if (VFS_ON_LIST(vfsp)) {
                gotlock = 1;
                vfs_list_lock();
        }
        vfs_clearmntopt_nolock(&vfsp->vfs_mntopts, opt, gotlock);
        if (gotlock)
                vfs_list_unlock();
}


/*
 * Set a mount option on.  If it’s not found in the table, it’s silently
 * ignored.  If the option has MO_IGNORE set, it is still set unless the
 * VFS_NOFORCEOPT bit is set in the flags.  Also, VFS_DISPLAY/VFS_NODISPLAY flag
 * bits can be used to toggle the MO_NODISPLAY bit for the option.
 * If the VFS_CREATEOPT flag bit is set then the first option slot with
 * MO_EMPTY set is created as the option passed in.
 *
 * The update_mnttab arg indicates whether mops is part of a vfs that is on
 * the vfs list.
 */
static void
vfs_setmntopt_nolock(mntopts_t *mops, const char *opt,
    const char *arg, int flags, int update_mnttab)
{
        mntopt_t *mop;
        uint_t i, count;
        char *sp;

        ASSERT(!update_mnttab || RW_WRITE_HELD(&vfslist));

        if (flags & VFS_CREATEOPT) {
                if (vfs_hasopt(mops, opt) != NULL) {
                        flags &= ~VFS_CREATEOPT;
                }
        }
        count = mops->mo_count;
        for (i = 0; i < count; i++) {
                mop = &mops->mo_list[i];

                if (mop->mo_flags & MO_EMPTY) {
                        if ((flags & VFS_CREATEOPT) == 0)
                                continue;
                        sp = kmem_alloc(strlen(opt) + 1, KM_SLEEP);
                        (void) strcpy(sp, opt);
                        mop->mo_name = sp;
                        if (arg != NULL)
                                mop->mo_flags = MO_HASVALUE;
                        else
                                mop->mo_flags = 0;
                } else if (strcmp(opt, mop->mo_name)) {
                        continue;
                }
                if ((mop->mo_flags & MO_IGNORE) && (flags & VFS_NOFORCEOPT))
                        break;
                if (arg != NULL && (mop->mo_flags & MO_HASVALUE) != 0) {
                        sp = kmem_alloc(strlen(arg) + 1, KM_SLEEP);
```

```
                        (void) strcpy(sp, arg);
                } else {
                        sp = NULL;
                }
                if (mop->mo_arg != NULL)
                        kmem_free(mop->mo_arg, strlen(mop->mo_arg) + 1);
                mop->mo_arg = sp;
                if (flags & VFS_DISPLAY)
                        mop->mo_flags &= ~MO_NODISPLAY;
                if (flags & VFS_NODISPLAY)
                        mop->mo_flags |= MO_NODISPLAY;
                mop->mo_flags |= MO_SET;
                if (mop->mo_cancel != NULL) {
                        char **cp;

                        for (cp = mop->mo_cancel; *cp != NULL; cp++)
                                vfs_clearmntopt_nolock(mops, *cp, 0);
                }
                if (update_mnttab)
                        vfs_mnttab_modtimeupd();
                break;
        }
}

void
vfs_setmntopt(struct vfs *vfsp, const char *opt, const char *arg, int flags)
{
        int gotlock = 0;

        if (VFS_ON_LIST(vfsp)) {
                gotlock = 1;
                vfs_list_lock();
        }
        vfs_setmntopt_nolock(&vfsp->vfs_mntopts, opt, arg, flags, gotlock);
        if (gotlock)
                vfs_list_unlock();
}


/*
 * Add a "tag" option to a mounted file system’s options list.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect mops.
 */
static mntopt_t *
vfs_addtag(mntopts_t *mops, const char *tag)
{
        uint_t count;
        mntopt_t *mop, *motbl;

        count = mops->mo_count + 1;
        motbl = kmem_zalloc(count * sizeof (mntopt_t), KM_SLEEP);
        if (mops->mo_count) {
                size_t len = (count - 1) * sizeof (mntopt_t);

                bcopy(mops->mo_list, motbl, len);
                kmem_free(mops->mo_list, len);
        }
        mops->mo_count = count;
        mops->mo_list = motbl;
        mop = &motbl[count - 1];
        mop->mo_flags = MO_TAG;
        mop->mo_name = kmem_alloc(strlen(tag) + 1, KM_SLEEP);
        (void) strcpy(mop->mo_name, tag);
        return (mop);
```

```
}

/*
 * Allow users to set arbitrary "tags" in a vfs’s mount options.
 * Broader use within the kernel is discouraged.
 */
int
vfs_settag(uint_t major, uint_t minor, const char *mntpt, const char *tag,
    cred_t *cr)
{
        vfs_t *vfsp;
        mntopts_t *mops;
        mntopt_t *mop;
        int found = 0;
        dev_t dev = makedevice(major, minor);
        int err = 0;
        char *buf = kmem_alloc(MAX_MNTOPT_STR, KM_SLEEP);

        /*
         * Find the desired mounted file system
         */
        vfs_list_lock();
        vfsp = rootvfs;
        do {
                if (vfsp->vfs_dev == dev &&
                    strcmp(mntpt, refstr_value(vfsp->vfs_mntpt)) == 0) {
                        found = 1;
                        break;
                }
                vfsp = vfsp->vfs_next;
        } while (vfsp != rootvfs);

        if (!found) {
                err = EINVAL;
                goto out;
        }
        err = secpolicy_fs_config(cr, vfsp);
        if (err != 0)
                goto out;

        mops = &vfsp->vfs_mntopts;
        /*
         * Add tag if it doesn’t already exist
         */
        if ((mop = vfs_hasopt(mops, tag)) == NULL) {
                int len;

                (void) vfs_buildoptionstr(mops, buf, MAX_MNTOPT_STR);
                len = strlen(buf);
                if (len + strlen(tag) + 2 > MAX_MNTOPT_STR) {
                        err = ENAMETOOLONG;
                        goto out;
                }
                mop = vfs_addtag(mops, tag);
        }
        if ((mop->mo_flags & MO_TAG) == 0) {
                err = EINVAL;
                goto out;
        }
        vfs_setmntopt_nolock(mops, tag, NULL, 0, 1);
out:
        vfs_list_unlock();
        kmem_free(buf, MAX_MNTOPT_STR);
        return (err);
}
```

```
/*
 * Allow users to remove arbitrary "tags" in a vfs’s mount options.
 * Broader use within the kernel is discouraged.
 */
int
vfs_clrtag(uint_t major, uint_t minor, const char *mntpt, const char *tag,
    cred_t *cr)
{
        vfs_t *vfsp;
        mntopt_t *mop;
        int found = 0;
        dev_t dev = makedevice(major, minor);
        int err = 0;

        /*
         * Find the desired mounted file system
         */
        vfs_list_lock();
        vfsp = rootvfs;
        do {
                if (vfsp->vfs_dev == dev &&
                    strcmp(mntpt, refstr_value(vfsp->vfs_mntpt)) == 0) {
                        found = 1;
                        break;
                }
                vfsp = vfsp->vfs_next;
        } while (vfsp != rootvfs);

        if (!found) {
                err = EINVAL;
                goto out;
        }
        err = secpolicy_fs_config(cr, vfsp);
        if (err != 0)
                goto out;

        if ((mop = vfs_hasopt(&vfsp->vfs_mntopts, tag)) == NULL) {
                err = EINVAL;
                goto out;
        }
        if ((mop->mo_flags & MO_TAG) == 0) {
                err = EINVAL;
                goto out;
        }
        vfs_clearmntopt_nolock(&vfsp->vfs_mntopts, tag, 1);
out:
        vfs_list_unlock();
        return (err);
}

/*
 * Function to parse an option string and fill in a mount options table.
 * Unknown options are silently ignored.  The input option string is modified
 * by replacing separators with nulls.  If the create flag is set, options
 * not found in the table are just added on the fly.  The table must have
 * an option slot marked MO_EMPTY to add an option on the fly.
 *
 * This function is *not* for general use by filesystems.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect mops..
 */
void
vfs_parsemntopts(mntopts_t *mops, char *osp, int create)
{
        char *s = osp, *p, *nextop, *valp, *cp, *ep;
        int setflg = VFS_NOFORCEOPT;

        if (osp == NULL)
                return;
        while (*s != ’\0’) {
                p = strchr(s, ’,’);     /* find next option */
                if (p == NULL) {
                        cp = NULL;
                        p = s + strlen(s);
                } else {
                        cp = p;         /* save location of comma */
                        *p++ = ’\0’;    /* mark end and point to next option */
                }
                nextop = p;
                p = strchr(s, ’=’);     /* look for value */
                if (p == NULL) {
                        valp = NULL;    /* no value supplied */
                } else {
                        ep = p;         /* save location of equals */
                        *p++ = ’\0’;    /* end option and point to value */
                        valp = p;
                }
                /*
                 * set option into options table
                 */
                if (create)
                        setflg |= VFS_CREATEOPT;
                vfs_setmntopt_nolock(mops, s, valp, setflg, 0);
                if (cp != NULL)
                        *cp = ’,’;      /* restore the comma */
                if (valp != NULL)
                        *ep = ’=’;      /* restore the equals */
                s = nextop;
        }
}

/*
 * Function to inquire if an option exists in a mount options table.
 * Returns a pointer to the option if it exists, else NULL.
 *
 * This function is *not* for general use by filesystems.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect mops.
 */
struct mntopt *
vfs_hasopt(const mntopts_t *mops, const char *opt)
{
        struct mntopt *mop;
        uint_t i, count;

        count = mops->mo_count;
        for (i = 0; i < count; i++) {
                mop = &mops->mo_list[i];

                if (mop->mo_flags & MO_EMPTY)
                        continue;
                if (strcmp(opt, mop->mo_name) == 0)
                        return (mop);
        }
        return (NULL);
}

/*
 * Function to inquire if an option is set in a mount options table.
 * Returns non-zero if set and fills in the arg pointer with a pointer to
```

```
 * the argument string or NULL if there is no argument string.
 */
static int
vfs_optionisset_nolock(const mntopts_t *mops, const char *opt, char **argp)
{
        struct mntopt *mop;
        uint_t i, count;

        count = mops->mo_count;
        for (i = 0; i < count; i++) {
                mop = &mops->mo_list[i];

                if (mop->mo_flags & MO_EMPTY)
                        continue;
                if (strcmp(opt, mop->mo_name))
                        continue;
                if ((mop->mo_flags & MO_SET) == 0)
                        return (0);
                if (argp != NULL && (mop->mo_flags & MO_HASVALUE) != 0)
                        *argp = mop->mo_arg;
                return (1);
        }
        return (0);
}


int
vfs_optionisset(const struct vfs *vfsp, const char *opt, char **argp)
{
        int ret;

        vfs_list_read_lock();
        ret = vfs_optionisset_nolock(&vfsp->vfs_mntopts, opt, argp);
        vfs_list_unlock();
        return (ret);
}


/*
 * Construct a comma separated string of the options set in the given
 * mount table, return the string in the given buffer.  Return non-zero if
 * the buffer would overflow.
 *
 * This function is *not* for general use by filesystems.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect mp.
 */
int
vfs_buildoptionstr(const mntopts_t *mp, char *buf, int len)
{
        char *cp;
        uint_t i;

        buf[0] = ’\0’;
        cp = buf;
        for (i = 0; i < mp->mo_count; i++) {
                struct mntopt *mop;

                mop = &mp->mo_list[i];
                if (mop->mo_flags & MO_SET) {
                        int optlen, comma = 0;

                        if (buf[0] != ’\0’)
                                comma = 1;
                        optlen = strlen(mop->mo_name);
                        if (strlen(buf) + comma + optlen + 1 > len)
                                goto err;
                        if (comma)
                                *cp++ = ',';
                        (void) strcpy(cp, mop->mo_name);
                        cp += optlen;
                        /*
                         * Append option value if there is one
                         */
                        if (mop->mo_arg != NULL) {
                                int arglen;

                                arglen = strlen(mop->mo_arg);
                                if (strlen(buf) + arglen + 2 > len)
                                        goto err;
                                *cp++ = '=';
                                (void) strcpy(cp, mop->mo_arg);
                                cp += arglen;
                        }
                }
        }
        return (0);
err:
        return (EOVERFLOW);
}

static void
vfs_freecancelopt(char **moc)
{
        if (moc != NULL) {
                int ccnt = 0;
                char **cp;

                for (cp = moc; *cp != NULL; cp++) {
                        kmem_free(*cp, strlen(*cp) + 1);
                        ccnt++;
                }
                kmem_free(moc, (ccnt + 1) * sizeof (char *));
        }
}

static void
vfs_freeopt(mntopt_t *mop)
{
        if (mop->mo_name != NULL)
                kmem_free(mop->mo_name, strlen(mop->mo_name) + 1);

        vfs_freecancelopt(mop->mo_cancel);

        if (mop->mo_arg != NULL)
                kmem_free(mop->mo_arg, strlen(mop->mo_arg) + 1);
}

/*
 * Free a mount options table
 *
 * This function is *not* for general use by filesystems.
 *
 * Note: caller is responsible for locking the vfs list, if needed,
 *       to protect mp.
 */
void
vfs_freeopttbl(mntopts_t *mp)
{
        uint_t i, count;
```

```
        count = mp->mo_count;
        for (i = 0; i < count; i++) {
                vfs_freeopt(&mp->mo_list[i]);
        }
        if (count) {
                kmem_free(mp->mo_list, sizeof (mntopt_t) * count);
                mp->mo_count = 0;
                mp->mo_list = NULL;
        }
}


/* ARGSUSED */
static int
vfs_mntdummyread(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cred,
        caller_context_t *ct)
{
        return (0);
}

/* ARGSUSED */
static int
vfs_mntdummywrite(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cred,
        caller_context_t *ct)
{
        return (0);
}

/*
 * The dummy vnode is currently used only by file events notification
 * module which is just interested in the timestamps.
 */
/* ARGSUSED */
static int
vfs_mntdummygetattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
    caller_context_t *ct)
{
        bzero(vap, sizeof (vattr_t));
        vap->va_type = VREG;
        vap->va_nlink = 1;
        vap->va_ctime = vfs_mnttab_ctime;
        /*
         * it is ok to just copy mtime as the time will be monotonically
         * increasing.
         */
        vap->va_mtime = vfs_mnttab_mtime;
        vap->va_atime = vap->va_mtime;
        return (0);
}

static void
vfs_mnttabvp_setup(void)
{
        vnode_t *tvp;
        vnodeops_t *vfs_mntdummyvnops;
        const fs_operation_def_t mnt_dummyvnodeops_template[] = {
                VOPNAME_READ,           { .vop_read = vfs_mntdummyread },
                VOPNAME_WRITE,          { .vop_write = vfs_mntdummywrite },
                VOPNAME_GETATTR,        { .vop_getattr = vfs_mntdummygetattr },
                VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
                NULL,                   NULL
        };

        if (vn_make_ops("mnttab", mnt_dummyvnodeops_template,
            &vfs_mntdummyvnops) != 0) {
                cmn_err(CE_WARN, "vfs_mnttabvp_setup: vn_make_ops failed");
                /* Shouldn’t happen, but not bad enough to panic */
                return;
        }

        /*
         * A global dummy vnode is allocated to represent mntfs files.
         * The mntfs file (/etc/mnttab) can be monitored for file events
         * and receive an event when mnttab changes. Dummy VOP calls
         * will be made on this vnode. The file events notification module
         * intercepts this vnode and delivers relevant events.
         */
        tvp = vn_alloc(KM_SLEEP);
        tvp->v_flag = VNOMOUNT|VNOMAP|VNOSWAP|VNOCACHE;
        vn_setops(tvp, vfs_mntdummyvnops);
        tvp->v_type = VREG;
        /*
         * The mnt dummy ops do not reference v_data.
         * No other module intercepting this vnode should either.
         * Just set it to point to itself.
         */
        tvp->v_data = (caddr_t)tvp;
        tvp->v_vfsp = rootvfs;
        vfs_mntdummyvp = tvp;
}

/*
 * performs fake read/write ops
 */
static void
vfs_mnttab_rwop(int rw)
{
        struct uio      uio;
        struct iovec    iov;
        char    buf[1];

        if (vfs_mntdummyvp == NULL)
                return;

        bzero(&uio, sizeof (uio));
        bzero(&iov, sizeof (iov));
        iov.iov_base = buf;
        iov.iov_len = 0;
        uio.uio_iov = &iov;
        uio.uio_iovcnt = 1;
        uio.uio_loffset = 0;
        uio.uio_segflg = UIO_SYSSPACE;
        uio.uio_resid = 0;
        if (rw) {
                (void) VOP_WRITE(vfs_mntdummyvp, &uio, 0, kcred, NULL);
        } else {
                (void) VOP_READ(vfs_mntdummyvp, &uio, 0, kcred, NULL);
        }
}

/*
 * Generate a write operation.
 */
void
vfs_mnttab_writeop(void)
{
        vfs_mnttab_rwop(1);
}

/*
 * Generate a read operation.
 */
```

```
void
vfs_mnttab_readop(void)
{
        vfs_mnttab_rwop(0);
}

/*
 * Free any mnttab information recorded in the vfs struct.
 * The vfs must not be on the vfs list.
 */
static void
vfs_freemnttab(struct vfs *vfsp)
{
        ASSERT(!VFS_ON_LIST(vfsp));

        /*
         * Free device and mount point information
         */
        if (vfsp->vfs_mntpt != NULL) {
                refstr_rele(vfsp->vfs_mntpt);
                vfsp->vfs_mntpt = NULL;
        }
        if (vfsp->vfs_resource != NULL) {
                refstr_rele(vfsp->vfs_resource);
                vfsp->vfs_resource = NULL;
        }
        /*
         * Now free mount options information
         */
        vfs_freeopttbl(&vfsp->vfs_mntopts);
}

/*
 * Return the last mnttab modification time
 */
void
vfs_mnttab_modtime(timespec_t *ts)
{
        ASSERT(RW_LOCK_HELD(&vfslist));
        *ts = vfs_mnttab_mtime;
}

/*
 * See if mnttab is changed
 */
void
vfs_mnttab_poll(timespec_t *old, struct pollhead **phpp)
{
        int changed;

        *phpp = (struct pollhead *)NULL;

        /*
         * Note: don’t grab vfs list lock before accessing vfs_mnttab_mtime.
         * Can lead to deadlock against vfs_mnttab_modtimeupd(). It is safe
         * to not grab the vfs list lock because tv_sec is monotonically
         * increasing.
         */

        changed = (old->tv_nsec != vfs_mnttab_mtime.tv_nsec) ||
            (old->tv_sec != vfs_mnttab_mtime.tv_sec);
        if (!changed) {
                *phpp = &vfs_pollhd;
        }
}
```

```
/* Provide a unique and monotonically-increasing timestamp. */
void
vfs_mono_time(timespec_t *ts)
{
        static volatile hrtime_t hrt;           /* The saved time. */
        hrtime_t        newhrt, oldhrt;         /* For effecting the CAS. */
        timespec_t      newts;

        /*
         * Try gethrestime() first, but be prepared to fabricate a sensible
         * answer at the first sign of any trouble.
         */
        gethrestime(&newts);
        newhrt = ts2hrt(&newts);
        for (;;) {
                oldhrt = hrt;
                if (newhrt <= hrt)
                        newhrt = hrt + 1;
                if (atomic_cas_64((uint64_t *)&hrt, oldhrt, newhrt) == oldhrt)
                        break;
        }
        hrt2ts(newhrt, ts);
}

/*
 * Update the mnttab modification time and wake up any waiters for
 * mnttab changes
 */
void
vfs_mnttab_modtimeupd()
{
        hrtime_t oldhrt, newhrt;

        ASSERT(RW_WRITE_HELD(&vfslist));
        oldhrt = ts2hrt(&vfs_mnttab_mtime);
        gethrestime(&vfs_mnttab_mtime);
        newhrt = ts2hrt(&vfs_mnttab_mtime);
        if (oldhrt == (hrtime_t)0)
                vfs_mnttab_ctime = vfs_mnttab_mtime;
        /*
         * Attempt to provide unique mtime (like uniqtime but not).
         */
        if (newhrt == oldhrt) {
                newhrt++;
                hrt2ts(newhrt, &vfs_mnttab_mtime);
        }
        pollwakeup(&vfs_pollhd, (short)POLLRDBAND);
        vfs_mnttab_writeop();
}

int
dounmount(struct vfs *vfsp, int flag, cred_t *cr)
{
        vnode_t *coveredvp;
        int error;
        extern void teardown_vopstats(vfs_t *);

        /*
         * Get covered vnode. This will be NULL if the vfs is not linked
         * into the file system name space (i.e., domount() with MNT_NOSPICE).
         */
        coveredvp = vfsp->vfs_vnodecovered;
        ASSERT(coveredvp == NULL || vn_vfswlock_held(coveredvp));

        /*
         * Purge all dnlc entries for this vfs.
         */
        (void) dnlc_purge_vfsp(vfsp, 0);

        /* For forcible umount, skip VFS_SYNC() since it may hang */
        if ((flag & MS_FORCE) == 0)
                (void) VFS_SYNC(vfsp, 0, cr);

        /*
         * Lock the vfs to maintain fs status quo during unmount.  This
         * has to be done after the sync because ufs_update tries to acquire
         * the vfs_reflock.
         */
        vfs_lock_wait(vfsp);

        if (error = VFS_UNMOUNT(vfsp, flag, cr)) {
                vfs_unlock(vfsp);
                if (coveredvp != NULL)
                        vn_vfsunlock(coveredvp);
        } else if (coveredvp != NULL) {
                teardown_vopstats(vfsp);
                /*
                 * vfs_remove() will do a VN_RELE(vfsp->vfs_vnodecovered)
                 * when it frees vfsp so we do a VN_HOLD() so we can
                 * continue to use coveredvp afterwards.
                 */
                VN_HOLD(coveredvp);
                vfs_remove(vfsp);
                vn_vfsunlock(coveredvp);
                VN_RELE(coveredvp);
        } else {
                teardown_vopstats(vfsp);
                /*
                 * Release the reference to vfs that is not linked
                 * into the name space.
                 */
                vfs_unlock(vfsp);
                VFS_RELE(vfsp);
        }
        return (error);
}


/*
 * Vfs_unmountall() is called by uadmin() to unmount all
 * mounted file systems (except the root file system) during shutdown.
 * It follows the existing locking protocol when traversing the vfs list
 * to sync and unmount vfses. Even though there should be no
 * other thread running while the system is shutting down, it is prudent
 * to still follow the locking protocol.
 */
void
vfs_unmountall(void)
{
        struct vfs *vfsp;
        struct vfs *prev_vfsp = NULL;
        int error;

        /*
         * Toss all dnlc entries now so that the per-vfs sync
         * and unmount operations don’t have to slog through
         * a bunch of uninteresting vnodes over and over again.
         */
        dnlc_purge();

        vfs_list_lock();
        for (vfsp = rootvfs->vfs_prev; vfsp != rootvfs; vfsp = prev_vfsp) {
                prev_vfsp = vfsp->vfs_prev;

                if (vfs_lock(vfsp) != 0)
                        continue;
                error = vn_vfswlock(vfsp->vfs_vnodecovered);
                vfs_unlock(vfsp);
                if (error)
                        continue;

                vfs_list_unlock();

                (void) VFS_SYNC(vfsp, SYNC_CLOSE, CRED());
                (void) dounmount(vfsp, 0, CRED());

                /*
                 * Since we dropped the vfslist lock above we must
                 * verify that next_vfsp still exists, else start over.
                 */
                vfs_list_lock();
                for (vfsp = rootvfs->vfs_prev;
                    vfsp != rootvfs; vfsp = vfsp->vfs_prev)
                        if (vfsp == prev_vfsp)
                                break;
                if (vfsp == rootvfs && prev_vfsp != rootvfs)
                        prev_vfsp = rootvfs->vfs_prev;
        }
        vfs_list_unlock();
}

/*
 * Called to add an entry to the end of the vfs mount in progress list
 */
void
vfs_addmip(dev_t dev, struct vfs *vfsp)
{
        struct ipmnt *mipp;

        mipp = (struct ipmnt *)kmem_alloc(sizeof (struct ipmnt), KM_SLEEP);
        mipp->mip_next = NULL;
        mipp->mip_dev = dev;
        mipp->mip_vfsp = vfsp;
        mutex_enter(&vfs_miplist_mutex);
        if (vfs_miplist_end != NULL)
                vfs_miplist_end->mip_next = mipp;
        else
                vfs_miplist = mipp;
        vfs_miplist_end = mipp;
        mutex_exit(&vfs_miplist_mutex);
}

/*
 * Called to remove an entry from the mount in progress list
 * Either because the mount completed or it failed.
 */
void
vfs_delmip(struct vfs *vfsp)
{
        struct ipmnt *mipp, *mipprev;

        mutex_enter(&vfs_miplist_mutex);
        mipprev = NULL;
        for (mipp = vfs_miplist;
            mipp && mipp->mip_vfsp != vfsp; mipp = mipp->mip_next) {
                mipprev = mipp;
        }
        if (mipp == NULL)
```

```
                return; /* shouldn’t happen */
        if (mipp == vfs_miplist_end)
                vfs_miplist_end = mipprev;
        if (mipprev == NULL)
                vfs_miplist = mipp->mip_next;
        else
                mipprev->mip_next = mipp->mip_next;
        mutex_exit(&vfs_miplist_mutex);
        kmem_free(mipp, sizeof (struct ipmnt));
}

/*
 * vfs_add is called by a specific filesystem’s mount routine to add
 * the new vfs into the vfs list/hash and to cover the mounted-on vnode.
 * The vfs should already have been locked by the caller.
 *
 * coveredvp is NULL if this is the root.
 */
void
vfs_add(vnode_t *coveredvp, struct vfs *vfsp, int mflag)
{
        int newflag;

        ASSERT(vfs_lock_held(vfsp));
        VFS_HOLD(vfsp);
        newflag = vfsp->vfs_flag;
        if (mflag & MS_RDONLY)
                newflag |= VFS_RDONLY;
        else
                newflag &= ~VFS_RDONLY;
        if (mflag & MS_NOSUID)
                newflag |= (VFS_NOSETUID|VFS_NODEVICES);
        else
                newflag &= ~(VFS_NOSETUID|VFS_NODEVICES);
        if (mflag & MS_NOMNTTAB)
                newflag |= VFS_NOMNTTAB;
        else
                newflag &= ~VFS_NOMNTTAB;

        if (coveredvp != NULL) {
                ASSERT(vn_vfswlock_held(coveredvp));
                coveredvp->v_vfsmountedhere = vfsp;
                VN_HOLD(coveredvp);
        }
        vfsp->vfs_vnodecovered = coveredvp;
        vfsp->vfs_flag = newflag;

        vfs_list_add(vfsp);
}

/*
 * Remove a vfs from the vfs list, null out the pointer from the
 * covered vnode to the vfs (v_vfsmountedhere), and null out the pointer
 * from the vfs to the covered vnode (vfs_vnodecovered). Release the
 * reference to the vfs and to the covered vnode.
 *
 * Called from dounmount after it’s confirmed with the file system
 * that the unmount is legal.
 */
void
vfs_remove(struct vfs *vfsp)
{
        vnode_t *vp;

        ASSERT(vfs_lock_held(vfsp));
        /*
         * Can’t unmount root.  Should never happen because fs will
         * be busy.
         */
        if (vfsp == rootvfs)
                panic("vfs_remove: unmounting root");

        vfs_list_remove(vfsp);

        /*
         * Unhook from the file system name space.
         */
        vp = vfsp->vfs_vnodecovered;
        ASSERT(vn_vfswlock_held(vp));
        vp->v_vfsmountedhere = NULL;
        vfsp->vfs_vnodecovered = NULL;
        VN_RELE(vp);

        /*
         * Release lock and wakeup anybody waiting.
         */
        vfs_unlock(vfsp);
        VFS_RELE(vfsp);
}

/*
 * Lock a filesystem to prevent access to it while mounting,
 * unmounting and syncing.  Return EBUSY immediately if lock
 * can’t be acquired.
 */
int
vfs_lock(vfs_t *vfsp)
{
        vn_vfslocks_entry_t *vpvfsentry;

        vpvfsentry = vn_vfslocks_getlock(vfsp);
        if (rwst_tryenter(&vpvfsentry->ve_lock, RW_WRITER))
                return (0);

        vn_vfslocks_rele(vpvfsentry);
        return (EBUSY);
}

int
vfs_rlock(vfs_t *vfsp)
{
        vn_vfslocks_entry_t *vpvfsentry;

        vpvfsentry = vn_vfslocks_getlock(vfsp);

        if (rwst_tryenter(&vpvfsentry->ve_lock, RW_READER))
                return (0);

        vn_vfslocks_rele(vpvfsentry);
        return (EBUSY);
}

void
vfs_lock_wait(vfs_t *vfsp)
{
        vn_vfslocks_entry_t *vpvfsentry;

        vpvfsentry = vn_vfslocks_getlock(vfsp);
        rwst_enter(&vpvfsentry->ve_lock, RW_WRITER);
}
```

```
void
vfs_rlock_wait(vfs_t *vfsp)
{
        vn_vfslocks_entry_t *vpvfsentry;

        vpvfsentry = vn_vfslocks_getlock(vfsp);
        rwst_enter(&vpvfsentry->ve_lock, RW_READER);
}

/*
 * Unlock a locked filesystem.
 */
void
vfs_unlock(vfs_t *vfsp)
{
        vn_vfslocks_entry_t *vpvfsentry;

        /*
         * vfs_unlock will mimic sema_v behaviour to fix 4748018.
         * And these changes should remain for the patch changes as it is.
         */
        if (panicstr)
                return;

        /*
         * ve_refcount needs to be dropped twice here.
         * 1. To release refernce after a call to vfs_locks_getlock()
         * 2. To release the reference from the locking routines like
         *    vfs_rlock_wait/vfs_wlock_wait/vfs_wlock etc,.
         */

        vpvfsentry = vn_vfslocks_getlock(vfsp);
        vn_vfslocks_rele(vpvfsentry);

        rwst_exit(&vpvfsentry->ve_lock);
        vn_vfslocks_rele(vpvfsentry);
}

/*
 * Utility routine that allows a filesystem to construct its
 * fsid in "the usual way" - by munging some underlying dev_t and
 * the filesystem type number into the 64-bit fsid.  Note that
 * this implicitly relies on dev_t persistence to make filesystem
 * id’s persistent.
 *
 * There’s nothing to prevent an individual fs from constructing its
 * fsid in a different way, and indeed they should.
 *
 * Since we want fsids to be 32-bit quantities (so that they can be
 * exported identically by either 32-bit or 64-bit APIs, as well as
 * the fact that fsid’s are "known" to NFS), we compress the device
 * number given down to 32-bits, and panic if that isn’t possible.
 */
void
vfs_make_fsid(fsid_t *fsi, dev_t dev, int val)
{
        if (!cmpldev((dev32_t *)&fsi->val[0], dev))
                panic("device number too big for fsid!");
        fsi->val[1] = val;
}

int
vfs_lock_held(vfs_t *vfsp)
{
        int held;
        vn_vfslocks_entry_t *vpvfsentry;
```

```c
        /*
         * vfs_lock_held will mimic sema_held behaviour
         * if panicstr is set. And these changes should remain
         * for the patch changes as it is.
         */
        if (panicstr)
                return (1);

        vpvfsentry = vn_vfslocks_getlock(vfsp);
        held = rwst_lock_held(&vpvfsentry->ve_lock, RW_WRITER);

        vn_vfslocks_rele(vpvfsentry);
        return (held);
}

struct _kthread *
vfs_lock_owner(vfs_t *vfsp)
{
        struct _kthread *owner;
        vn_vfslocks_entry_t *vpvfsentry;

        /*
         * vfs_wlock_held will mimic sema_held behaviour
         * if panicstr is set. And these changes should remain
         * for the patch changes as it is.
         */
        if (panicstr)
                return (NULL);

        vpvfsentry = vn_vfslocks_getlock(vfsp);
        owner = rwst_owner(&vpvfsentry->ve_lock);

        vn_vfslocks_rele(vpvfsentry);
        return (owner);
}

/*
 * vfs list locking.
 *
 * Rather than manipulate the vfslist lock directly, we abstract into lock
 * and unlock routines to allow the locking implementation to be changed for
 * clustering.
 *
 * Whenever the vfs list is modified through its hash links, the overall list
 * lock must be obtained before locking the relevant hash bucket.  But to see
 * whether a given vfs is on the list, it suffices to obtain the lock for the
 * hash bucket without getting the overall list lock.  (See getvfs() below.)
 */

void
vfs_list_lock()
{
        rw_enter(&vfslist, RW_WRITER);
}

void
vfs_list_read_lock()
{
        rw_enter(&vfslist, RW_READER);
}

void
vfs_list_unlock()
{
        rw_exit(&vfslist);
```

```c
}

/*
 * Low level worker routines for adding entries to and removing entries from
 * the vfs list.
 */

static void
vfs_hash_add(struct vfs *vfsp, int insert_at_head)
{
        int vhno;
        struct vfs **hp;
        dev_t dev;

        ASSERT(RW_WRITE_HELD(&vfslist));

        dev = expldev(vfsp->vfs_fsid.val[0]);
        vhno = VFSHASH(getmajor(dev), getminor(dev));

        mutex_enter(&rvfs_list[vhno].rvfs_lock);

        /*
         * Link into the hash table, inserting it at the end, so that LOFS
         * with the same fsid as UFS (or other) file systems will not hide the
         * UFS.
         */
        if (insert_at_head) {
                vfsp->vfs_hash = rvfs_list[vhno].rvfs_head;
                rvfs_list[vhno].rvfs_head = vfsp;
        } else {
                for (hp = &rvfs_list[vhno].rvfs_head; *hp != NULL;
                    hp = &(*hp)->vfs_hash)
                        continue;
                /*
                 * hp now contains the address of the pointer to update
                 * to effect the insertion.
                 */
                vfsp->vfs_hash = NULL;
                *hp = vfsp;
        }

        rvfs_list[vhno].rvfs_len++;
        mutex_exit(&rvfs_list[vhno].rvfs_lock);
}


static void
vfs_hash_remove(struct vfs *vfsp)
{
        int vhno;
        struct vfs *tvfsp;
        dev_t dev;

        ASSERT(RW_WRITE_HELD(&vfslist));

        dev = expldev(vfsp->vfs_fsid.val[0]);
        vhno = VFSHASH(getmajor(dev), getminor(dev));

        mutex_enter(&rvfs_list[vhno].rvfs_lock);

        /*
         * Remove from hash.
         */
        if (rvfs_list[vhno].rvfs_head == vfsp) {
                rvfs_list[vhno].rvfs_head = vfsp->vfs_hash;
                rvfs_list[vhno].rvfs_len--;
                goto foundit;
        }
        for (tvfsp = rvfs_list[vhno].rvfs_head; tvfsp != NULL;
            tvfsp = tvfsp->vfs_hash) {
                if (tvfsp->vfs_hash == vfsp) {
                        tvfsp->vfs_hash = vfsp->vfs_hash;
                        rvfs_list[vhno].rvfs_len--;
                        goto foundit;
                }
        }
        cmn_err(CE_WARN, "vfs_list_remove: vfs not found in hash");

foundit:

        mutex_exit(&rvfs_list[vhno].rvfs_lock);
}


void
vfs_list_add(struct vfs *vfsp)
{
        zone_t *zone;

        /*
         * Typically, the vfs_t will have been created on behalf of the file
         * system in vfs_init, where it will have been provided with a
         * vfs_impl_t. This, however, might be lacking if the vfs_t was created
         * by an unbundled file system. We therefore check for such an example
         * before stamping the vfs_t with its creation time for the benefit of
         * mntfs.
         */
        if (vfsp->vfs_implp == NULL)
                vfsimpl_setup(vfsp);
        vfs_mono_time(&vfsp->vfs_hrctime);

        /*
         * The zone that owns the mount is the one that performed the mount.
         * Note that this isn't necessarily the same as the zone mounted into.
         * The corresponding zone_rele_ref() will be done when the vfs_t
         * is being free'd.
         */
        vfsp->vfs_zone = curproc->p_zone;
        zone_init_ref(&vfsp->vfs_implp->vi_zone_ref);
        zone_hold_ref(vfsp->vfs_zone, &vfsp->vfs_implp->vi_zone_ref,
            ZONE_REF_VFS);

        /*
         * Find the zone mounted into, and put this mount on its vfs list.
         */
        zone = zone_find_by_path(refstr_value(vfsp->vfs_mntpt));
        ASSERT(zone != NULL);
        /*
         * Special casing for the root vfs.  This structure is allocated
         * statically and hooked onto rootvfs at link time.  During the
         * vfs_mountroot call at system startup time, the root file system's
         * VFS_MOUNTROOT routine will call vfs_add with this root vfs struct
         * as argument.  The code below must detect and handle this special
         * case.  The only apparent justification for this special casing is
         * to ensure that the root file system appears at the head of the
         * list.
         *
         * XXX: I'm assuming that it's ok to do normal list locking when
         *      adding the entry for the root file system (this used to be
         *      done with no locks held).
         */
        vfs_list_lock();
```

```
        /*
         * Link into the vfs list proper.
         */
        if (vfsp == &root) {
                /*
                 * Assert: This vfs is already on the list as its first entry.
                 * Thus, there's nothing to do.
                 */
                ASSERT(rootvfs == vfsp);
                /*
                 * Add it to the head of the global zone's vfslist.
                 */
                ASSERT(zone == global_zone);
                ASSERT(zone->zone_vfslist == NULL);
                zone->zone_vfslist = vfsp;
        } else {
                /*
                 * Link to end of list using vfs_prev (as rootvfs is now a
                 * doubly linked circular list) so list is in mount order for
                 * mnttab use.
                 */
                rootvfs->vfs_prev->vfs_next = vfsp;
                vfsp->vfs_prev = rootvfs->vfs_prev;
                rootvfs->vfs_prev = vfsp;
                vfsp->vfs_next = rootvfs;

                /*
                 * Do it again for the zone-private list (which may be NULL).
                 */
                if (zone->zone_vfslist == NULL) {
                        ASSERT(zone != global_zone);
                        zone->zone_vfslist = vfsp;
                } else {
                        zone->zone_vfslist->vfs_zone_prev->vfs_zone_next = vfsp;
                        vfsp->vfs_zone_prev = zone->zone_vfslist->vfs_zone_prev;
                        zone->zone_vfslist->vfs_zone_prev = vfsp;
                        vfsp->vfs_zone_next = zone->zone_vfslist;
                }
        }

        /*
         * Link into the hash table, inserting it at the end, so that LOFS
         * with the same fsid as UFS (or other) file systems will not hide
         * the UFS.
         */
        vfs_hash_add(vfsp, 0);

        /*
         * Link into tree indexed by mntpoint, for vfs_mntpoint2vfsp
         * mntix discerns entries with the same key
         */
        vfsp->vfs_mntix = ++vfs_curr_mntix;
        avl_add(&vfs_by_dev, vfsp);

        /*
         * Link into tree indexed by dev, for vfs_devismounted
         */
        avl_add(&vfs_by_mntpnt, vfsp);

        /*
#endif /* ! codereview */
         * update the mnttab modification time
         */
        vfs_mnttab_modtimeupd();
        vfs_list_unlock();
        zone_rele(zone);
```

```
}

void
vfs_list_remove(struct vfs *vfsp)
{
        zone_t *zone;

        zone = zone_find_by_path(refstr_value(vfsp->vfs_mntpt));
        ASSERT(zone != NULL);
        /*
         * Callers are responsible for preventing attempts to unmount the
         * root.
         */
        ASSERT(vfsp != rootvfs);

        vfs_list_lock();

        /*
         * Remove from avl trees
         */
        avl_remove(&vfs_by_mntpnt, vfsp);
        avl_remove(&vfs_by_dev, vfsp);

        /*
#endif /* ! codereview */
         * Remove from hash.
         */
        vfs_hash_remove(vfsp);

        /*
         * Remove from vfs list.
         */
        vfsp->vfs_prev->vfs_next = vfsp->vfs_next;
        vfsp->vfs_next->vfs_prev = vfsp->vfs_prev;
        vfsp->vfs_next = vfsp->vfs_prev = NULL;

        /*
         * Remove from zone-specific vfs list.
         */
        if (zone->zone_vfslist == vfsp)
                zone->zone_vfslist = vfsp->vfs_zone_next;

        if (vfsp->vfs_zone_next == vfsp) {
                ASSERT(vfsp->vfs_zone_prev == vfsp);
                ASSERT(zone->zone_vfslist == vfsp);
                zone->zone_vfslist = NULL;
        }

        vfsp->vfs_zone_prev->vfs_zone_next = vfsp->vfs_zone_next;
        vfsp->vfs_zone_next->vfs_zone_prev = vfsp->vfs_zone_prev;
        vfsp->vfs_zone_next = vfsp->vfs_zone_prev = NULL;

        /*
         * update the mnttab modification time
         */
        vfs_mnttab_modtimeupd();
        vfs_list_unlock();
        zone_rele(zone);
}

struct vfs *
getvfs(fsid_t *fsid)
{
        struct vfs *vfsp;
        int val0 = fsid->val[0];
        int val1 = fsid->val[1];
```

```
        dev_t dev = expldev(val0);
        int vhno = VFSHASH(getmajor(dev), getminor(dev));
        kmutex_t *hmp = &rvfs_list[vhno].rvfs_lock;

        mutex_enter(hmp);
        for (vfsp = rvfs_list[vhno].rvfs_head; vfsp; vfsp = vfsp->vfs_hash) {
                if (vfsp->vfs_fsid.val[0] == val0 &&
                    vfsp->vfs_fsid.val[1] == val1) {
                        VFS_HOLD(vfsp);
                        mutex_exit(hmp);
                        return (vfsp);
                }
        }
        mutex_exit(hmp);
        return (NULL);
}

/*
 * Search the vfs mount in progress list for a specified device/vfs entry.
 * Returns 0 if the first entry in the list that the device matches has the
 * given vfs pointer as well.  If the device matches but a different vfs
 * pointer is encountered in the list before the given vfs pointer then
 * a 1 is returned.
 */

int
vfs_devmounting(dev_t dev, struct vfs *vfsp)
{
        int retval = 0;
        struct ipmnt *mipp;

        mutex_enter(&vfs_miplist_mutex);
        for (mipp = vfs_miplist; mipp != NULL; mipp = mipp->mip_next) {
                if (mipp->mip_dev == dev) {
                        if (mipp->mip_vfsp != vfsp)
                                retval = 1;
                        break;
                }
        }
        mutex_exit(&vfs_miplist_mutex);
        return (retval);
}

/*
 * Search the vfs list for a specified device.  Returns 1, if entry is found
 * or 0 if no suitable entry is found.
 */

int
vfs_devismounted(dev_t dev)
{
        struct vfs *vfsp;
        int found = 0;
        struct vfs search;
        avl_index_t index;

        search.vfs_dev = dev;
        search.vfs_mntix = 0;
        int found;

        vfs_list_read_lock();

        /*
         * there might be several entries with the same dev in the tree,
         * only discerned by mntix. To find the first, we start with a mntix
         * of 0. The search will fail. The following avl_nearest will give
         * us the actual first entry.
         */
        VERIFY(avl_find(&vfs_by_dev, &search, &index) == NULL);
        vfsp = avl_nearest(&vfs_by_dev, index, AVL_AFTER);

        if (vfsp != NULL && vfsp->vfs_dev == dev)
        vfsp = rootvfs;
        found = 0;
        do {
                if (vfsp->vfs_dev == dev) {
                found = 1;
                        break;
                }
                vfsp = vfsp->vfs_next;
        } while (vfsp != rootvfs);

        vfs_list_unlock();
        return (found);
}

/*
 * Search the vfs list for a specified device.  Returns a pointer to it
 * or NULL if no suitable entry is found. The caller of this routine
 * is responsible for releasing the returned vfs pointer.
 */
struct vfs *
vfs_dev2vfsp(dev_t dev)
{
        struct vfs *vfsp;
        int found;
        struct vfs search;
        avl_index_t index;

        search.vfs_dev = dev;
        search.vfs_mntix = 0;
#endif /* ! codereview */

        vfs_list_read_lock();

        /*
         * there might be several entries with the same dev in the tree,
         * only discerned by mntix. To find the first, we start with a mntix
         * of 0. The search will fail. The following avl_nearest will give
         * us the actual first entry.
         */
        VERIFY(avl_find(&vfs_by_dev, &search, &index) == NULL);
        vfsp = avl_nearest(&vfs_by_dev, index, AVL_AFTER);

        vfsp = rootvfs;
        found = 0;
        while (vfsp != NULL && vfsp->vfs_dev == dev) {
        do {
                /*
                 * The following could be made more efficient by making
                 * the entire loop use vfs_zone_next if the call is from
                 * a zone.  The only callers, however, ustat(2) and
                 * umount2(2), don't seem to justify the added
                 * complexity at present.
                 */
                if (ZONE_PATH_VISIBLE(refstr_value(vfsp->vfs_mntpt),
                if (vfsp->vfs_dev == dev &&
                    ZONE_PATH_VISIBLE(refstr_value(vfsp->vfs_mntpt),
                    curproc->p_zone)) {
                        VFS_HOLD(vfsp);
                        found = 1;
                        break;
```

```
                }
                vfsp = AVL_NEXT(&vfs_by_dev, vfsp);
        }
                vfsp = vfsp->vfs_next;
        } while (vfsp != rootvfs);
        vfs_list_unlock();
        return (found ? vfsp : NULL);
        return (found ? vfsp: NULL);
}

/*
 * Search the vfs list for a specified mntpoint.  Returns a pointer to it
 * or NULL if no suitable entry is found. The caller of this routine
 * is responsible for releasing the returned vfs pointer.
 *
 * Note that if multiple mntpoints match, the last one matching is
 * returned in an attempt to return the "top" mount when overlay
 * mounts are covering the same mount point.  This is accomplished by starting
 * at the end of the list and working our way backwards, stopping at the first
 * matching mount.
 */
struct vfs *
vfs_mntpoint2vfsp(const char *mp)
{
        struct vfs *vfsp;
        struct vfs *retvfsp = NULL;
        zone_t *zone = curproc->p_zone;
        struct vfs *list;

        vfs_list_read_lock();
        if (getzoneid() == GLOBAL_ZONEID) {
                /*
                 * The global zone may see filesystems in any zone.
                 */
                struct vfs search;
                search.vfs_mntpt = refstr_alloc(mp);
                search.vfs_mntix = UINT64_MAX;
                avl_index_t index;

                /*
                 * there might be several entries with the same mntpnt in the
                 * tree, only discerned by mntix. To find the last, we start
                 * with a mntix of UINT64_MAX. The search will fail. The
                 * following avl_nearest will give  us the actual last entry
                 * matching the mntpnt.
                 */
                VERIFY(avl_find(&vfs_by_mntpnt, &search, &index) == 0);
                vfsp = avl_nearest(&vfs_by_mntpnt, index, AVL_BEFORE);

                refstr_rele(search.vfs_mntpt);

                if (vfsp != NULL &&
                    strcmp(refstr_value(vfsp->vfs_mntpt), mp) == 0)
                vfsp = rootvfs->vfs_prev;
                do {
                        if (strcmp(refstr_value(vfsp->vfs_mntpt), mp) == 0) {
                        retvfsp = vfsp;
                                break;
                        }
                        vfsp = vfsp->vfs_prev;
                } while (vfsp != rootvfs->vfs_prev);
        } else if ((list = zone->zone_vfslist) != NULL) {
                const char *mntpt;

                vfsp = list->vfs_zone_prev;
                do {
                        mntpt = refstr_value(vfsp->vfs_mntpt);
                        mntpt = ZONE_PATH_TRANSLATE(mntpt, zone);
                        if (strcmp(mntpt, mp) == 0) {
                                retvfsp = vfsp;
                                break;
                        }
                        vfsp = vfsp->vfs_zone_prev;
                } while (vfsp != list->vfs_zone_prev);
        }
        if (retvfsp)
                VFS_HOLD(retvfsp);
        vfs_list_unlock();
        return (retvfsp);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   21396 Mon Sep 28 19:41:48 2015
new/usr/src/uts/common/sys/vfs.h
6265 speed up mount/umount
**********************************************************
______unchanged_portion_omitted_
```

```
extern avl_tree_t       vskstat_tree;
extern kmutex_t         vskstat_tree_lock;

/*
 * Structure per mounted file system.  Each mounted file system has
 * an array of operations and an instance record.
 *
 * The file systems are kept on a doubly linked circular list headed by
 * "rootvfs".
 * File system implementations should not access this list;
 * it's intended for use only in the kernel's vfs layer.
 *
 * Each zone also has its own list of mounts, containing filesystems mounted
 * somewhere within the filesystem tree rooted at the zone's rootpath.  The
 * list is doubly linked to match the global list.
 *
 * mnttab locking: the in-kernel mnttab uses the vfs_mntpt, vfs_resource and
 * vfs_mntopts fields in the vfs_t. mntpt and resource are refstr_ts that
 * are set at mount time and can only be modified during a remount.
 * It is safe to read these fields if you can prevent a remount on the vfs,
 * or through the convenience funcs vfs_getmntpoint() and vfs_getresource().
 * The mntopts field may only be accessed through the provided convenience
 * functions, as it is protected by the vfs list lock. Modifying a mount
 * option requires grabbing the vfs list write lock, which can be a very
 * high latency lock.
 */
struct zone;            /* from zone.h */
struct fem_head;        /* from fem.h */

typedef struct vfs {
        struct vfs      *vfs_next;              /* next VFS in VFS list */
        struct vfs      *vfs_prev;              /* prev VFS in VFS list */
        avl_node_t      vfs_avldev;             /* by dev index */
        avl_node_t      vfs_avlmntpnt;          /* by mntpnt index */
        /*
         * global mount count to define an order on entries in
         * the avl trees with same dev/mountpoint
         */
        uint64_t        vfs_mntix;
#endif /* ! codereview */

/* vfs_op should not be used directly.  Accessor functions are provided */
        vfsops_t        *vfs_op;                /* operations on VFS */

        struct vnode    *vfs_vnodecovered;      /* vnode mounted on */
        uint_t          vfs_flag;               /* flags */
        uint_t          vfs_bsize;              /* native block size */
        int             vfs_fstype;             /* file system type index */
        fsid_t          vfs_fsid;               /* file system id */
        void            *vfs_data;              /* private data */
        dev_t           vfs_dev;                /* device of mounted VFS */
        ulong_t         vfs_bcount;             /* I/O count (accounting) */
        struct vfs      *vfs_list;              /* sync list pointer */
        struct vfs      *vfs_hash;              /* hash list pointer */
        ksema_t         vfs_reflock;            /* mount/unmount/sync lock */
        uint_t          vfs_count;              /* vfs reference count */
        mntopts_t       vfs_mntopts;            /* options mounted with */
        refstr_t        *vfs_resource;          /* mounted resource name */
        refstr_t        *vfs_mntpt;             /* mount point name */
```

```
        time_t          vfs_mtime;              /* time we were mounted */
        struct vfs_impl *vfs_implp;             /* impl specific data */
        /*
         * Zones support.  Note that the zone that "owns" the mount isn't
         * necessarily the same as the zone in which the zone is visible.
         * That is, vfs_zone and (vfs_zone_next|vfs_zone_prev) may refer to
         * different zones.
         */
        struct zone     *vfs_zone;              /* zone that owns the mount */
        struct vfs      *vfs_zone_next;         /* next VFS visible in zone */
        struct vfs      *vfs_zone_prev;         /* prev VFS visible in zone */

        struct fem_head *vfs_femhead;           /* fs monitoring */
        minor_t         vfs_lofi_minor;         /* minor if lofi mount */
} vfs_t;

#define vfs_featureset  vfs_implp->vi_featureset
#define vfs_vskap       vfs_implp->vi_vskap
#define vfs_fstypevsp   vfs_implp->vi_fstypevsp
#define vfs_vopstats    vfs_implp->vi_vopstats
#define vfs_hrctime     vfs_implp->vi_hrctime

/*
 * VFS flags.
 */
#define VFS_RDONLY      0x01            /* read-only vfs */
#define VFS_NOMNTTAB    0x02            /* vfs not seen in mnttab */
#define VFS_NOSETUID    0x08            /* setuid disallowed */
#define VFS_REMOUNT     0x10            /* modify mount options only */
#define VFS_NOTRUNC     0x20            /* does not truncate long file names */
#define VFS_UNLINKABLE  0x40            /* unlink(2) can be applied to root */
#define VFS_PXFS        0x80            /* clustering: global fs proxy vfs */
#define VFS_UNMOUNTED   0x100           /* file system has been unmounted */
#define VFS_NBMAND      0x200           /* allow non-blocking mandatory locks */
#define VFS_XATTR       0x400           /* fs supports extended attributes */
#define VFS_NODEVICES   0x800           /* device-special files disallowed */
#define VFS_NOEXEC      0x1000          /* executables disallowed */
#define VFS_STATS       0x2000          /* file system can collect stats */
#define VFS_XID         0x4000          /* file system supports extended ids */

#define VFS_NORESOURCE  "unspecified_resource"
#define VFS_NOMNTPT     "unspecified_mountpoint"

/*
 * VFS features are implemented as bits set in the vfs_t.
 * The vfs_feature_t typedef is a 64-bit number that will translate
 * into an element in an array of bitmaps and a bit in that element.
 * Developers must not depend on the implementation of this and
 * need to use vfs_has_feature()/vfs_set_feature() routines.
 */
typedef uint64_t        vfs_feature_t;

#define VFSFT_XVATTR            0x100000001     /* Supports xvattr for attrs */
#define VFSFT_CASEINSENSITIVE   0x100000002     /* Supports case-insensitive */
#define VFSFT_NOCASESENSITIVE   0x100000004     /* NOT case-sensitive */
#define VFSFT_DIRENTFLAGS       0x100000008     /* Supports dirent flags */
#define VFSFT_ACLONCREATE       0x100000010     /* Supports ACL on create */
#define VFSFT_ACEMASKONACCESS   0x100000020     /* Can use ACEMASK for access */
#define VFSFT_SYSATTR_VIEWS     0x100000040     /* Supports sysattr view i/f */
#define VFSFT_ACCESS_FILTER     0x100000080     /* dirents filtered by access */
#define VFSFT_REPARSE           0x100000100     /* Supports reparse point */
#define VFSFT_ZEROCOPY_SUPPORTED        0x100000200
                                /* Support loaning /returning cache buffer */
/*
 * Argument structure for mount(2).
 *
 * Flags are defined in <sys/mount.h>.
 *
 * Note that if the MS_SYSSPACE bit is set in flags, the pointer fields in
 * this structure are to be interpreted as kernel addresses.  File systems
 * should be prepared for this possibility.
 */
struct mounta {
        char    *spec;
        char    *dir;
        int     flags;
        char    *fstype;
        char    *dataptr;
        int     datalen;
        char    *optptr;
        int     optlen;
};

/*
 * Reasons for calling the vfs_mountroot() operation.
 */
enum whymountroot { ROOT_INIT, ROOT_REMOUNT, ROOT_UNMOUNT};
typedef enum whymountroot whymountroot_t;

/*
 * Reasons for calling the VFS_VNSTATE():
 */
enum vntrans {
        VNTRANS_EXISTS,
        VNTRANS_IDLED,
        VNTRANS_RECLAIMED,
        VNTRANS_DESTROYED
};
typedef enum vntrans vntrans_t;

/*
 * VFS_OPS defines all the vfs operations.  It is used to define
 * the vfsops structure (below) and the fs_func_p union (vfs_opreg.h).
 */
#define VFS_OPS                                                         \
        int     (*vfs_mount)(vfs_t *, vnode_t *, struct mounta *, cred_t *); \
        int     (*vfs_unmount)(vfs_t *, int, cred_t *);                 \
        int     (*vfs_root)(vfs_t *, vnode_t **);                       \
        int     (*vfs_statvfs)(vfs_t *, statvfs64_t *);                 \
        int     (*vfs_sync)(vfs_t *, short, cred_t *);                  \
        int     (*vfs_vget)(vfs_t *, vnode_t **, fid_t *);              \
        int     (*vfs_mountroot)(vfs_t *, enum whymountroot);           \
        void    (*vfs_freevfs)(vfs_t *);                                \
        int     (*vfs_vnstate)(vfs_t *, vnode_t *, vntrans_t)   /* NB: No ";" */

/*
 * Operations supported on virtual file system.
 */
struct vfsops {
        VFS_OPS;        /* Signature of all vfs operations (vfsops) */
};

extern int      fsop_mount(vfs_t *, vnode_t *, struct mounta *, cred_t *);
extern int      fsop_unmount(vfs_t *, int, cred_t *);
extern int      fsop_root(vfs_t *, vnode_t **);
extern int      fsop_statfs(vfs_t *, statvfs64_t *);
extern int      fsop_sync(vfs_t *, short, cred_t *);
extern int      fsop_vget(vfs_t *, vnode_t **, fid_t *);
extern int      fsop_mountroot(vfs_t *, enum whymountroot);
extern void     fsop_freefs(vfs_t *);
extern int      fsop_sync_by_kind(int, short, cred_t *);
extern int      fsop_vnstate(vfs_t *, vnode_t *, vntrans_t);
```

```
#define VFS_MOUNT(vfsp, mvp, uap, cr) fsop_mount(vfsp, mvp, uap, cr)
#define VFS_UNMOUNT(vfsp, flag, cr) fsop_unmount(vfsp, flag, cr)
#define VFS_ROOT(vfsp, vpp) fsop_root(vfsp, vpp)
#define VFS_STATVFS(vfsp, sp) fsop_statfs(vfsp, sp)
#define VFS_SYNC(vfsp, flag, cr) fsop_sync(vfsp, flag, cr)
#define VFS_VGET(vfsp, vpp, fidp) fsop_vget(vfsp, vpp, fidp)
#define VFS_MOUNTROOT(vfsp, init) fsop_mountroot(vfsp, init)
#define VFS_FREEVFS(vfsp) fsop_freefs(vfsp)
#define VFS_VNSTATE(vfsp, vn, ns)       fsop_vnstate(vfsp, vn, ns)

#define VFSNAME_MOUNT           "mount"
#define VFSNAME_UNMOUNT         "unmount"
#define VFSNAME_ROOT            "root"
#define VFSNAME_STATVFS         "statvfs"
#define VFSNAME_SYNC            "sync"
#define VFSNAME_VGET            "vget"
#define VFSNAME_MOUNTROOT       "mountroot"
#define VFSNAME_FREEVFS         "freevfs"
#define VFSNAME_VNSTATE         "vnstate"
/*
 * Filesystem type switch table.
 */

typedef struct vfssw {
        char            *vsw_name;      /* type name -- max len _ST_FSTYPSZ */
        int             (*vsw_init) (int, char *);
                                /* init routine (for non-loadable fs only) */
        int             vsw_flag;       /* flags */
        mntopts_t       vsw_optproto;   /* mount options table prototype */
        uint_t          vsw_count;      /* count of references */
        kmutex_t        vsw_lock;       /* lock to protect vsw_count */
        vfsops_t        vsw_vfsops;     /* filesystem operations vector */
} vfssw_t;

/*
 * Filesystem type definition record.  All file systems must export a record
 * of this type through their modlfs structure.  N.B., changing the version
 * number requires a change in sys/modctl.h.
 */

typedef struct vfsdef_v5 {
        int             def_version;    /* structure version, must be first */
        char            *name;          /* filesystem type name */
        int             (*init) (int, char *);  /* init routine */
        int             flags;          /* filesystem flags */
        mntopts_t       *optproto;      /* mount options table prototype */
} vfsdef_v5;

typedef struct vfsdef_v5 vfsdef_t;

enum {
        VFSDEF_VERSION = 5
};

/*
 * flags for vfssw and vfsdef
 */
#define VSW_HASPROTO    0x01    /* struct has a mount options prototype */
#define VSW_CANRWRO     0x02    /* file system can transition from rw to ro */
#define VSW_CANREMOUNT  0x04    /* file system supports remounts */
#define VSW_NOTZONESAFE 0x08    /* zone_enter(2) should fail for these files */
#define VSW_VOLATILEDEV 0x10    /* vfs_dev can change each time fs is mounted */
#define VSW_STATS       0x20    /* file system can collect stats */
#define VSW_XID         0x40    /* file system supports extended ids */
#define VSW_CANLOFI     0x80    /* file system supports lofi mounts */
```

```
#define VSW_ZMOUNT      0x100   /* file system always allowed in a zone */

#define VSW_INSTALLED   0x8000  /* this vsw is associated with a file system */

/*
 * A flag for vfs_setpath().
 */
#define VFSSP_VERBATIM  0x1     /* do not prefix the supplied path */

#if defined(_KERNEL) || defined(_FAKE_KERNEL)

/*
 * Private vfs data, NOT to be used by a file system implementation.
 */

#define VFS_FEATURE_MAXSZ       4

typedef struct vfs_impl {
        /* Counted array - Bitmap of vfs features */
        uint32_t        vi_featureset[VFS_FEATURE_MAXSZ];
        /*
         * Support for statistics on the vnode operations
         */
        vsk_anchor_t    *vi_vskap;              /* anchor for vopstats’ kstat */
        vopstats_t      *vi_fstypevsp;          /* ptr to per-fstype vopstats */
        vopstats_t      vi_vopstats;            /* per-mount vnode op stats */

        timespec_t      vi_hrctime;             /* High-res creation time */

        zone_ref_t      vi_zone_ref;            /* reference to zone */
} vfs_impl_t;

/*
 * Public operations.
 */
struct umounta;
struct statvfsa;
struct fstatvfsa;

void    vfs_freevfsops(vfsops_t *);
int     vfs_freevfsops_by_type(int);
void    vfs_setops(vfs_t *, vfsops_t *);
vfsops_t *vfs_getops(vfs_t *vfsp);
int     vfs_matchops(vfs_t *, vfsops_t *);
int     vfs_can_sync(vfs_t *vfsp);
vfs_t   *vfs_alloc(int);
void    vfs_free(vfs_t *);
void    vfs_init(vfs_t *vfsp, vfsops_t *, void *);
void    vfsimpl_setup(vfs_t *vfsp);
void    vfsimpl_teardown(vfs_t *vfsp);
void    vn_exists(vnode_t *);
void    vn_idle(vnode_t *);
void    vn_reclaim(vnode_t *);
void    vn_invalid(vnode_t *);

int     rootconf(void);
int     svm_rootconf(void);
int     domount(char *, struct mounta *, vnode_t *, struct cred *,
            struct vfs **);
int     dounmount(struct vfs *, int, cred_t *);
int     vfs_lock(struct vfs *);
int     vfs_rlock(struct vfs *);
void    vfs_lock_wait(struct vfs *);
void    vfs_rlock_wait(struct vfs *);
void    vfs_unlock(struct vfs *);
int     vfs_lock_held(struct vfs *);
```

```
struct  _kthread *vfs_lock_owner(struct vfs *);
void    sync(void);
void    vfs_sync(int);
void    vfs_mountroot(void);
void    vfs_add(vnode_t *, struct vfs *, int);
void    vfs_remove(struct vfs *);

/* VFS feature routines */
void    vfs_set_feature(vfs_t *, vfs_feature_t);
void    vfs_clear_feature(vfs_t *, vfs_feature_t);
int     vfs_has_feature(vfs_t *, vfs_feature_t);
void    vfs_propagate_features(vfs_t *, vfs_t *);

/* The following functions are not for general use by filesystems */

void    vfs_createopttbl(mntopts_t *, const char *);
void    vfs_copyopttbl(const mntopts_t *, mntopts_t *);
void    vfs_mergeopttbl(const mntopts_t *, const mntopts_t *, mntopts_t *);
void    vfs_freeopttbl(mntopts_t *);
void    vfs_parsemntopts(mntopts_t *, char *, int);
int     vfs_buildoptionstr(const mntopts_t *, char *, int);
struct mntopt *vfs_hasopt(const mntopts_t *, const char *);
void    vfs_mnttab_modtimeupd(void);

void    vfs_clearmntopt(struct vfs *, const char *);
void    vfs_setmntopt(struct vfs *, const char *, const char *, int);
void    vfs_setresource(struct vfs *, const char *, uint32_t);
void    vfs_setmntpoint(struct vfs *, const char *, uint32_t);
refstr_t *vfs_getresource(const struct vfs *);
refstr_t *vfs_getmntpoint(const struct vfs *);
int     vfs_optionisset(const struct vfs *, const char *, char **);
int     vfs_settag(uint_t, uint_t, const char *, const char *, cred_t *);
int     vfs_clrtag(uint_t, uint_t, const char *, const char *, cred_t *);
void    vfs_syncall(void);
void    vfs_syncprogress(void);
void    vfsinit(void);
void    vfs_unmountall(void);
void    vfs_make_fsid(fsid_t *, dev_t, int);
void    vfs_addmip(dev_t, struct vfs *);
void    vfs_delmip(struct vfs *);
int     vfs_devismounted(dev_t);
int     vfs_devmounting(dev_t, struct vfs *);
int     vfs_opsinuse(vfsops_t *);
struct vfs *getvfs(fsid_t *);
struct vfs *vfs_dev2vfsp(dev_t);
struct vfs *vfs_mntpoint2vfsp(const char *);
struct vfssw *allocate_vfssw(const char *);
struct vfssw *vfs_getvfssw(const char *);
struct vfssw *vfs_getvfsswbyname(const char *);
struct vfssw *vfs_getvfsswbyvfsops(vfsops_t *);
void    vfs_refvfssw(struct vfssw *);
void    vfs_unrefvfssw(struct vfssw *);
uint_t  vf_to_stf(uint_t);
void    vfs_mnttab_modtime(timespec_t *);
void    vfs_mnttab_poll(timespec_t *, struct pollhead **);

void    vfs_list_lock(void);
void    vfs_list_read_lock(void);
void    vfs_list_unlock(void);
void    vfs_list_add(struct vfs *);
void    vfs_list_remove(struct vfs *);
void    vfs_hold(vfs_t *vfsp);
void    vfs_rele(vfs_t *vfsp);
void    fs_freevfs(vfs_t *);
void    vfs_root_redev(vfs_t *vfsp, dev_t ndev, int fstype);
```

```
int     vfs_zone_change_safe(vfs_t *);

int     vfs_get_lofi(vfs_t *, vnode_t **);

#define VFSHASH(maj, min) (((int)((maj)+(min))) & (vfshsz - 1))
#define VFS_ON_LIST(vfsp) \
        ((vfsp)->vfs_next != (vfsp) && (vfsp)->vfs_next != NULL)

/*
 * Globals.
 */

extern struct vfssw vfssw[];            /* table of filesystem types */
extern krwlock_t vfssw_lock;
extern char rootfstype[];               /* name of root fstype */
extern const int nfstype;               /* # of elements in vfssw array */
extern vfsops_t *EIO_vfsops;            /* operations for vfs being torn-down */

/*
 * The following variables are private to the the kernel's vfs layer.  File
 * system implementations should not access them.
 */
extern struct vfs *rootvfs;             /* ptr to root vfs structure */
typedef struct {
        struct vfs *rvfs_head;          /* head vfs in chain */
        kmutex_t rvfs_lock;             /* mutex protecting this chain */
        uint32_t rvfs_len;              /* length of this chain */
} rvfs_t;
extern rvfs_t *rvfs_list;
extern int vfshsz;                      /* # of elements in rvfs_head array */
extern const mntopts_t vfs_mntopts;     /* globally recognized options */

#endif /* defined(_KERNEL) */

#define VFS_HOLD(vfsp) { \
        vfs_hold(vfsp); \
}

#define VFS_RELE(vfsp)  { \
        vfs_rele(vfsp); \
}

#define VFS_INIT(vfsp, op, data) { \
        vfs_init((vfsp), (op), (data)); \
}


#define VFS_INSTALLED(vfsswp)   (((vfsswp)->vsw_flag & VSW_INSTALLED) != 0)
#define ALLOCATED_VFSSW(vswp)           ((vswp)->vsw_name[0] != '\0')
#define RLOCK_VFSSW()                   (rw_enter(&vfssw_lock, RW_READER))
#define RUNLOCK_VFSSW()                 (rw_exit(&vfssw_lock))
#define WLOCK_VFSSW()                   (rw_enter(&vfssw_lock, RW_WRITER))
#define WUNLOCK_VFSSW()                 (rw_exit(&vfssw_lock))
#define VFSSW_LOCKED()                  (RW_LOCK_HELD(&vfssw_lock))
#define VFSSW_WRITE_LOCKED()            (RW_WRITE_HELD(&vfssw_lock))
/*
 * VFS_SYNC flags.
 */
#define SYNC_ATTR       0x01            /* sync attributes only */
#define SYNC_CLOSE      0x02            /* close open file */
#define SYNC_ALL        0x04            /* force to sync all fs */

#ifdef  __cplusplus
}
#endif
```

```
#endif  /* _SYS_VFS_H */
```