```
**********************************************************
  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 */
```

```
128 /*
129  * VFS global data.
130  */
131 vnode_t *rootdir;               /* pointer to root inode vnode. */
132 vnode_t *devicesdir;            /* pointer to inode of devices root */
133 vnode_t *devdir;                /* pointer to inode of dev root */

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

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

158 /*
159  * Table for generic options recognized in the VFS layer and acted
160  * on at this level before parsing file system specific options.
161  * The nosuid option is stronger than any of the devices and setuid
162  * options, so those are canceled when nosuid is seen.
163  *
164  * All options which are added here need to be added to the
165  * list of standard options in usr/src/cmd/fs.d/fslib.c as well.
166  */
167 /*
168  * VFS Mount options table
169  */
170 static char *ro_cancel[] = { MNTOPT_RW, NULL };
171 static char *rw_cancel[] = { MNTOPT_RO, NULL };
172 static char *suid_cancel[] = { MNTOPT_NOSUID, NULL };
173 static char *nosuid_cancel[] = { MNTOPT_SUID, MNTOPT_DEVICES, MNTOPT_NODEVICES,
174     MNTOPT_NOSETUID, MNTOPT_SETUID, NULL };
175 static char *devices_cancel[] = { MNTOPT_NODEVICES, NULL };
176 static char *nodevices_cancel[] = { MNTOPT_DEVICES, NULL };
177 static char *setuid_cancel[] = { MNTOPT_NOSETUID, NULL };
178 static char *nosetuid_cancel[] = { MNTOPT_SETUID, NULL };
179 static char *nbmand_cancel[] = { MNTOPT_NONBMAND, NULL };
180 static char *nonbmand_cancel[] = { MNTOPT_NBMAND, NULL };
181 static char *exec_cancel[] = { MNTOPT_NOEXEC, NULL };
182 static char *noexec_cancel[] = { MNTOPT_EXEC, NULL };

184 static const mntopt_t mntopts[] = {
185 /*
186  *      option name             cancel options          default arg     flags
187  */
188         { MNTOPT_REMOUNT,       NULL,                   NULL,
189                 MO_NODISPLAY, (void *)0 },
190         { MNTOPT_RO,            ro_cancel,              NULL,           0,
191                 (void *)0 },
192         { MNTOPT_RW,            rw_cancel,              NULL,           0,
193                 (void *)0 },
```



```
194         { MNTOPT_SUID,          suid_cancel,            NULL,           0,
195                 (void *)0 },
196         { MNTOPT_NOSUID,        nosuid_cancel,          NULL,           0,
197                 (void *)0 },
198         { MNTOPT_DEVICES,       devices_cancel,         NULL,           0,
199                 (void *)0 },
200         { MNTOPT_NODEVICES,     nodevices_cancel,       NULL,           0,
201                 (void *)0 },
202         { MNTOPT_SETUID,        setuid_cancel,          NULL,           0,
203                 (void *)0 },
204         { MNTOPT_NOSETUID,      nosetuid_cancel,        NULL,           0,
205                 (void *)0 },
206         { MNTOPT_NBMAND,        nbmand_cancel,          NULL,           0,
207                 (void *)0 },
208         { MNTOPT_NONBMAND,      nonbmand_cancel,        NULL,           0,
209                 (void *)0 },
210         { MNTOPT_EXEC,          exec_cancel,            NULL,           0,
211                 (void *)0 },
212         { MNTOPT_NOEXEC,        noexec_cancel,          NULL,           0,
213                 (void *)0 },
214 };

216 const mntopts_t vfs_mntopts = {
217         sizeof (mntopts) / sizeof (mntopt_t),
218         (mntopt_t *)&mntopts[0]
219 };

221 /*
222  * File system operation dispatch functions.
223  */

225 int
226 fsop_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
227 {
228         return (*(vfsp)->vfs_op->vfs_mount)(vfsp, mvp, uap, cr);
229 }

231 int
232 fsop_unmount(vfs_t *vfsp, int flag, cred_t *cr)
233 {
234         return (*(vfsp)->vfs_op->vfs_unmount)(vfsp, flag, cr);
235 }

237 int
238 fsop_root(vfs_t *vfsp, vnode_t **vpp)
239 {
240         refstr_t *mntpt;
241         int ret = (*(vfsp)->vfs_op->vfs_root)(vfsp, vpp);
242         /*
243          * Make sure this root has a path.  With lofs, it is possible to have
244          * a NULL mountpoint.
245          */
246         if (ret == 0 && vfsp->vfs_mntpt != NULL && (*vpp)->v_path == NULL) {
247                 mntpt = vfs_getmntpoint(vfsp);
248                 vn_setpath_str(*vpp, refstr_value(mntpt),
249                     strlen(refstr_value(mntpt)));
250                 refstr_rele(mntpt);
251         }

253         return (ret);
254 }

256 int
257 fsop_statfs(vfs_t *vfsp, statvfs64_t *sp)
258 {
259         return (*(vfsp)->vfs_op->vfs_statvfs)(vfsp, sp);
```

```
}

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)
```



```
{
        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)
```

```
392                 return (error);

394         vfssw[fstype].vsw_flag |= VSW_INSTALLED;

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

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

405         return (0);
406 }

408 int
409 vfs_makefsops(const fs_operation_def_t *template, vfsops_t **actual)
410 {
411         int error;
412         int unused_ops;

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

416         error = fs_copyfsops(template, *actual, &unused_ops);
417         if (error != 0) {
418                 kmem_free(*actual, sizeof (vfsops_t));
419                 *actual = NULL;
420                 return (error);
421         }

423         return (0);
424 }

426 /*
427  * Free a vfsops structure created as a result of vfs_makefsops().
428  * NOTE: For a vfsops structure initialized by vfs_setfsops(), use
429  * vfs_freevfsops_by_type().
430  */
431 void
432 vfs_freevfsops(vfsops_t *vfsops)
433 {
434         kmem_free(vfsops, sizeof (vfsops_t));
435 }

437 /*
438  * Since the vfsops structure is part of the vfssw table and wasn't
439  * really allocated, we're not really freeing anything.  We keep
440  * the name for consistency with vfs_freevfsops().  We do, however,
441  * need to take care of a little bookkeeping.
442  * NOTE: For a vfsops structure created by vfs_setfsops(), use
443  * vfs_freevfsops_by_type().
444  */
445 int
446 vfs_freevfsops_by_type(int fstype)
447 {

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

453         WLOCK_VFSSW();
454         if ((vfssw[fstype].vsw_flag & VSW_INSTALLED) == 0) {
455                 WUNLOCK_VFSSW();
456                 return (EINVAL);
457         }
```



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

462         return (0);
463 }

465 /* Support routines used to reference vfs_op */

467 /* Set the operations vector for a vfs */
468 void
469 vfs_setops(vfs_t *vfsp, vfsops_t *vfsops)
470 {
471         vfsops_t        *op;

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

476         op = vfsp->vfs_op;
477         membar_consumer();
478         if (vfsp->vfs_femhead == NULL &&
479             atomic_cas_ptr(&vfsp->vfs_op, op, vfsops) == op) {
480                 return;
481         }
482         fsem_setvfsops(vfsp, vfsops);
483 }

485 /* Retrieve the operations vector for a vfs */
486 vfsops_t *
487 vfs_getops(vfs_t *vfsp)
488 {
489         vfsops_t        *op;

491         ASSERT(vfsp != NULL);

493         op = vfsp->vfs_op;
494         membar_consumer();
495         if (vfsp->vfs_femhead == NULL && op == vfsp->vfs_op) {
496                 return (op);
497         } else {
498                 return (fsem_getvfsops(vfsp));
499         }
500 }

502 /*
503  * Returns non-zero (1) if the vfsops matches that of the vfs.
504  * Returns zero (0) if not.
505  */
506 int
507 vfs_matchops(vfs_t *vfsp, vfsops_t *vfsops)
508 {
509         return (vfs_getops(vfsp) == vfsops);
510 }

512 /*
513  * Returns non-zero (1) if the file system has installed a non-default,
514  * non-error vfs_sync routine.  Returns zero (0) otherwise.
515  */
516 int
517 vfs_can_sync(vfs_t *vfsp)
518 {
519         /* vfs_sync() routine is not the default/error function */
520         return (vfs_getops(vfsp)->vfs_sync != fs_sync);
521 }

523 /*
```

```
 * 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.
 */

/*
```



```
 * 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)
```

```
656                 return (1);
657         return (0);
658 }

660 /*
661 #endif /* ! codereview */
662  * External routines.
663  */

665 krwlock_t vfssw_lock;   /* lock accesses to vfssw */

667 /*
668  * Lock for accessing the vfs linked list.  Initialized in vfs_mountroot(),
669  * but otherwise should be accessed only via vfs_list_lock() and
670  * vfs_list_unlock().  Also used to protect the timestamp for mods to the list.
671  */
672 static krwlock_t vfslist;

674 /*
675  * Mount devfs on /devices. This is done right after root is mounted
676  * to provide device access support for the system
677  */
678 static void
679 vfs_mountdevices(void)
680 {
681         struct vfssw *vsw;
682         struct vnode *mvp;
683         struct mounta mounta = {        /* fake mounta for devfs_mount() */
684                 NULL,
685                 NULL,
686                 MS_SYSSPACE,
687                 NULL,
688                 NULL,
689                 0,
690                 NULL,
691                 0
692         };

694         /*
695          * _init devfs module to fill in the vfssw
696          */
697         if (modload("fs", "devfs") == -1)
698                 panic("Cannot _init devfs module");

700         /*
701          * Hold vfs
702          */
703         RLOCK_VFSSW();
704         vsw = vfs_getvfsswbyname("devfs");
705         VFS_INIT(&devices, &vsw->vsw_vfsops, NULL);
706         VFS_HOLD(&devices);

708         /*
709          * Locate mount point
710          */
711         if (lookupname("/devices", UIO_SYSSPACE, FOLLOW, NULLVPP, &mvp))
712                 panic("Cannot find /devices");

714         /*
715          * Perform the mount of /devices
716          */
717         if (VFS_MOUNT(&devices, mvp, &mounta, CRED()))
718                 panic("Cannot mount /devices");

720         RUNLOCK_VFSSW();
```



```
722         /*
723          * Set appropriate members and add to vfs list for mnttab display
724          */
725         vfs_setresource(&devices, "/devices", 0);
726         vfs_setmntpoint(&devices, "/devices", 0);

728         /*
729          * Hold the root of /devices so it won't go away
730          */
731         if (VFS_ROOT(&devices, &devicesdir))
732                 panic("vfs_mountdevices: not devices root");

734         if (vfs_lock(&devices) != 0) {
735                 VN_RELE(devicesdir);
736                 cmn_err(CE_NOTE, "Cannot acquire vfs_lock of /devices");
737                 return;
738         }

740         if (vn_vfswlock(mvp) != 0) {
741                 vfs_unlock(&devices);
742                 VN_RELE(devicesdir);
743                 cmn_err(CE_NOTE, "Cannot acquire vfswlock of /devices");
744                 return;
745         }

747         vfs_add(mvp, &devices, 0);
748         vn_vfsunlock(mvp);
749         vfs_unlock(&devices);
750         VN_RELE(devicesdir);
751 }

753 /*
754  * mount the first instance of /dev  to root and remain mounted
755  */
756 static void
757 vfs_mountdev1(void)
758 {
759         struct vfssw *vsw;
760         struct vnode *mvp;
761         struct mounta mounta = {        /* fake mounta for sdev_mount() */
762                 NULL,
763                 NULL,
764                 MS_SYSSPACE | MS_OVERLAY,
765                 NULL,
766                 NULL,
767                 0,
768                 NULL,
769                 0
770         };

772         /*
773          * _init dev module to fill in the vfssw
774          */
775         if (modload("fs", "dev") == -1)
776                 cmn_err(CE_PANIC, "Cannot _init dev module\n");

778         /*
779          * Hold vfs
780          */
781         RLOCK_VFSSW();
782         vsw = vfs_getvfsswbyname("dev");
783         VFS_INIT(&dev, &vsw->vsw_vfsops, NULL);
784         VFS_HOLD(&dev);

786         /*
787          * Locate mount point
```

```
788          */
789         if (lookupname("/dev", UIO_SYSSPACE, FOLLOW, NULLVPP, &mvp))
790                 cmn_err(CE_PANIC, "Cannot find /dev\n");

792         /*
793          * Perform the mount of /dev
794          */
795         if (VFS_MOUNT(&dev, mvp, &mounta, CRED()))
796                 cmn_err(CE_PANIC, "Cannot mount /dev 1\n");

798         RUNLOCK_VFSSW();

800         /*
801          * Set appropriate members and add to vfs list for mnttab display
802          */
803         vfs_setresource(&dev, "/dev", 0);
804         vfs_setmntpoint(&dev, "/dev", 0);

806         /*
807          * Hold the root of /dev so it won't go away
808          */
809         if (VFS_ROOT(&dev, &devdir))
810                 cmn_err(CE_PANIC, "vfs_mountdev1: not dev root");

812         if (vfs_lock(&dev) != 0) {
813                 VN_RELE(devdir);
814                 cmn_err(CE_NOTE, "Cannot acquire vfs_lock of /dev");
815                 return;
816         }

818         if (vn_vfswlock(mvp) != 0) {
819                 vfs_unlock(&dev);
820                 VN_RELE(devdir);
821                 cmn_err(CE_NOTE, "Cannot acquire vfswlock of /dev");
822                 return;
823         }

825         vfs_add(mvp, &dev, 0);
826         vn_vfsunlock(mvp);
827         vfs_unlock(&dev);
828         VN_RELE(devdir);
829 }

831 /*
832  * Mount required filesystem. This is done right after root is mounted.
833  */
834 static void
835 vfs_mountfs(char *module, char *spec, char *path)
836 {
837         struct vnode *mvp;
838         struct mounta mounta;
839         vfs_t *vfsp;

841         mounta.flags = MS_SYSSPACE | MS_DATA;
842         mounta.fstype = module;
843         mounta.spec = spec;
844         mounta.dir = path;
845         if (lookupname(path, UIO_SYSSPACE, FOLLOW, NULLVPP, &mvp)) {
846                 cmn_err(CE_WARN, "Cannot find %s", path);
847                 return;
848         }
849         if (domount(NULL, &mounta, mvp, CRED(), &vfsp))
850                 cmn_err(CE_WARN, "Cannot mount %s", path);
851         else
852                 VFS_RELE(vfsp);
853         VN_RELE(mvp);
```



```
854 }

856 /*
857  * vfs_mountroot is called by main() to mount the root filesystem.
858  */
859 void
860 vfs_mountroot(void)
861 {
862         struct vnode    *rvp = NULL;
863         char            *path;
864         size_t          plen;
865         struct vfssw    *vswp;
866         proc_t          *p;

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

871         /*
872          * Alloc the avl trees for quick indexing via dev and mountpoint
873          */
874         avl_create(&vfs_by_dev, vfs_cmp_dev, sizeof(vfs_t),
875             offsetof(vfs_t, vfs_avldev));
876         avl_create(&vfs_by_mntpnt, vfs_cmp_mntpnt, sizeof(vfs_t),
877             offsetof(vfs_t, vfs_avlmntpnt));

879         /*
880 #endif /* ! codereview */
881          * Alloc the vfs hash bucket array and locks
882          */
883         rvfs_list = kmem_zalloc(vfshsz * sizeof (rvfs_t), KM_SLEEP);

885         /*
886          * Call machine-dependent routine "rootconf" to choose a root
887          * file system type.
888          */
889         if (rootconf())
890                 panic("vfs_mountroot: cannot mount root");
891         /*
892          * Get vnode for '/'.  Set up rootdir, u.u_rdir and u.u_cdir
893          * to point to it.  These are used by lookuppn() so that it
894          * knows where to start from ('/' or '.').
895          */
896         vfs_setmntpoint(rootvfs, "/", 0);
897         if (VFS_ROOT(rootvfs, &rootdir))
898                 panic("vfs_mountroot: no root vnode");

900         /*
901          * At this point, the process tree consists of p0 and possibly some
902          * direct children of p0.  (i.e. there are no grandchildren)
903          *
904          * Walk through them all, setting their current directory.
905          */
906         mutex_enter(&pidlock);
907         for (p = practive; p != NULL; p = p->p_next) {
908                 ASSERT(p == &p0 || p->p_parent == &p0);

910                 PTOU(p)->u_cdir = rootdir;
911                 VN_HOLD(PTOU(p)->u_cdir);
912                 PTOU(p)->u_rdir = NULL;
913         }
914         mutex_exit(&pidlock);

916         /*
917          * Setup the global zone's rootvp, now that it exists.
918          */
919         global_zone->zone_rootvp = rootdir;
```

```
920         VN_HOLD(global_zone->zone_rootvp);
921
922         /*
923          * Notify the module code that it can begin using the
924          * root filesystem instead of the boot program's services.
925          */
926         modrootloaded = 1;
927
928         /*
929          * Special handling for a ZFS root file system.
930          */
931         zfs_boot_init();
932
933         /*
934          * Set up mnttab information for root
935          */
936         vfs_setresource(rootvfs, rootfs.bo_name, 0);
937
938         /*
939          * Notify cluster software that the root filesystem is available.
940          */
941         clboot_mountroot();
942
943         /* Now that we're all done with the root FS, set up its vopstats */
944         if ((vswp = vfs_getvfsswbyvfsops(vfs_getops(rootvfs))) != NULL) {
945                 /* Set flag for statistics collection */
946                 if (vswp->vsw_flag & VSW_STATS) {
947                         initialize_vopstats(&rootvfs->vfs_vopstats);
948                         rootvfs->vfs_flag |= VFS_STATS;
949                         rootvfs->vfs_fstypevsp =
950                             get_fstype_vopstats(rootvfs, vswp);
951                         rootvfs->vfs_vskap = get_vskstat_anchor(rootvfs);
952                 }
953                 vfs_unrefvfssw(vswp);
954         }
955
956         /*
957          * Mount /devices, /dev instance 1, /system/contract, /etc/mnttab,
958          * /etc/svc/volatile, /etc/dfs/sharetab, /system/object, and /proc.
959          */
960         vfs_mountdevices();
961         vfs_mountdev1();
962
963         vfs_mountfs("ctfs", "ctfs", CTFS_ROOT);
964         vfs_mountfs("proc", "/proc", "/proc");
965         vfs_mountfs("mntfs", "/etc/mnttab", "/etc/mnttab");
966         vfs_mountfs("tmpfs", "/etc/svc/volatile", "/etc/svc/volatile");
967         vfs_mountfs("objfs", "objfs", OBJFS_ROOT);
968
969         if (getzoneid() == GLOBAL_ZONEID) {
970                 vfs_mountfs("sharefs", "sharefs", "/etc/dfs/sharetab");
971         }
972
973 #ifdef __sparc
974         /*
975          * This bit of magic can go away when we convert sparc to
976          * the new boot architecture based on ramdisk.
977          *
978          * Booting off a mirrored root volume:
979          * At this point, we have booted and mounted root on a
980          * single component of the mirror.  Complete the boot
981          * by configuring SVM and converting the root to the
982          * dev_t of the mirrored root device.  This dev_t conversion
983          * only works because the underlying device doesn't change.
984          */
985         if (root_is_svm) {
```



```
986                 if (svm_rootconf()) {
987                         panic("vfs_mountroot: cannot remount root");
988                 }
989
990                 /*
991                  * mnttab should reflect the new root device
992                  */
993                 vfs_lock_wait(rootvfs);
994                 vfs_setresource(rootvfs, rootfs.bo_name, 0);
995                 vfs_unlock(rootvfs);
996         }
997 #endif /* __sparc */
998
999         if (strcmp(rootfs.bo_fstype, "zfs") != 0) {
1000                /*
1001                 * Look up the root device via devfs so that a dv_node is
1002                 * created for it. The vnode is never VN_RELE()ed.
1003                 * We allocate more than MAXPATHLEN so that the
1004                 * buffer passed to i_ddi_prompath_to_devfspath() is
1005                 * exactly MAXPATHLEN (the function expects a buffer
1006                 * of that length).
1007                 */
1008                plen = strlen("/devices");
1009                path = kmem_alloc(plen + MAXPATHLEN, KM_SLEEP);
1010                (void) strcpy(path, "/devices");
1011
1012                if (i_ddi_prompath_to_devfspath(rootfs.bo_name, path + plen)
1013                    != DDI_SUCCESS ||
1014                    lookupname(path, UIO_SYSSPACE, FOLLOW, NULLVPP, &rvp)) {
1015
1016                        /* NUL terminate in case "path" has garbage */
1017                        path[plen + MAXPATHLEN - 1] = '\0';
1018 #ifdef  DEBUG
1019                        cmn_err(CE_WARN, "!Cannot lookup root device: %s",
1020                            path);
1021 #endif
1022                }
1023                kmem_free(path, plen + MAXPATHLEN);
1024        }
1025
1026        vfs_mnttabvp_setup();
1027 }
1028
1029 /*
1030  * Check to see if our "block device" is actually a file.  If so,
1031  * automatically add a lofi device, and keep track of this fact.
1032  */
1033 static int
1034 lofi_add(const char *fsname, struct vfs *vfsp,
1035     mntopts_t *mntopts, struct mounta *uap)
1036 {
1037         int fromspace = (uap->flags & MS_SYSSPACE) ?
1038             UIO_SYSSPACE : UIO_USERSPACE;
1039         struct lofi_ioctl *li = NULL;
1040         struct vnode *vp = NULL;
1041         struct pathname pn = { NULL };
1042         ldi_ident_t ldi_id;
1043         ldi_handle_t ldi_hdl;
1044         vfssw_t *vfssw;
1045         int minor;
1046         int err = 0;
1047
1048         if ((vfssw = vfs_getvfssw(fsname)) == NULL)
1049                 return (0);
1050
1051         if (!(vfssw->vsw_flag & VSW_CANLOFI)) {
```

```
1052                 vfs_unrefvfssw(vfssw);
1053                 return (0);
1054         }

1056         vfs_unrefvfssw(vfssw);
1057         vfssw = NULL;

1059         if (pn_get(uap->spec, fromspace, &pn) != 0)
1060                 return (0);

1062         if (lookupname(uap->spec, fromspace, FOLLOW, NULL, &vp) != 0)
1063                 goto out;

1065         if (vp->v_type != VREG)
1066                 goto out;

1068         /* OK, this is a lofi mount. */

1070         if ((uap->flags & (MS_REMOUNT|MS_GLOBAL)) ||
1071             vfs_optionisset_nolock(mntopts, MNTOPT_SUID, NULL) ||
1072             vfs_optionisset_nolock(mntopts, MNTOPT_SETUID, NULL) ||
1073             vfs_optionisset_nolock(mntopts, MNTOPT_DEVICES, NULL)) {
1074                 err = EINVAL;
1075                 goto out;
1076         }

1078         ldi_id = ldi_ident_from_anon();
1079         li = kmem_zalloc(sizeof (*li), KM_SLEEP);
1080         (void) strlcpy(li->li_filename, pn.pn_path, MAXPATHLEN);

1082         err = ldi_open_by_name("/dev/lofictl", FREAD | FWRITE, kcred,
1083             &ldi_hdl, ldi_id);

1085         if (err)
1086                 goto out2;

1088         err = ldi_ioctl(ldi_hdl, LOFI_MAP_FILE, (intptr_t)li,
1089             FREAD | FWRITE | FKIOCTL, kcred, &minor);

1091         (void) ldi_close(ldi_hdl, FREAD | FWRITE, kcred);

1093         if (!err)
1094                 vfsp->vfs_lofi_minor = minor;

1096 out2:
1097         ldi_ident_release(ldi_id);
1098 out:
1099         if (li != NULL)
1100                 kmem_free(li, sizeof (*li));
1101         if (vp != NULL)
1102                 VN_RELE(vp);
1103         pn_free(&pn);
1104         return (err);
1105 }

1107 static void
1108 lofi_remove(struct vfs *vfsp)
1109 {
1110         struct lofi_ioctl *li = NULL;
1111         ldi_ident_t ldi_id;
1112         ldi_handle_t ldi_hdl;
1113         int err;

1115         if (vfsp->vfs_lofi_minor == 0)
1116                 return;
```



```
1118         ldi_id = ldi_ident_from_anon();

1120         li = kmem_zalloc(sizeof (*li), KM_SLEEP);
1121         li->li_minor = vfsp->vfs_lofi_minor;
1122         li->li_cleanup = B_TRUE;

1124         err = ldi_open_by_name("/dev/lofictl", FREAD | FWRITE, kcred,
1125             &ldi_hdl, ldi_id);

1127         if (err)
1128                 goto out;

1130         err = ldi_ioctl(ldi_hdl, LOFI_UNMAP_FILE_MINOR, (intptr_t)li,
1131             FREAD | FWRITE | FKIOCTL, kcred, NULL);

1133         (void) ldi_close(ldi_hdl, FREAD | FWRITE, kcred);

1135         if (!err)
1136                 vfsp->vfs_lofi_minor = 0;

1138 out:
1139         ldi_ident_release(ldi_id);
1140         if (li != NULL)
1141                 kmem_free(li, sizeof (*li));
1142 }

1144 /*
1145  * Common mount code.  Called from the system call entry point, from autofs,
1146  * nfsv4 trigger mounts, and from pxfs.
1147  *
1148  * Takes the effective file system type, mount arguments, the mount point
1149  * vnode, flags specifying whether the mount is a remount and whether it
1150  * should be entered into the vfs list, and credentials.  Fills in its vfspp
1151  * parameter with the mounted file system instance's vfs.
1152  *
1153  * Note that the effective file system type is specified as a string.  It may
1154  * be null, in which case it's determined from the mount arguments, and may
1155  * differ from the type specified in the mount arguments; this is a hook to
1156  * allow interposition when instantiating file system instances.
1157  *
1158  * The caller is responsible for releasing its own hold on the mount point
1159  * vp (this routine does its own hold when necessary).
1160  * Also note that for remounts, the mount point vp should be the vnode for
1161  * the root of the file system rather than the vnode that the file system
1162  * is mounted on top of.
1163  */
1164 int
1165 domount(char *fsname, struct mounta *uap, vnode_t *vp, struct cred *credp,
1166         struct vfs **vfspp)
1167 {
1168         struct vfssw    *vswp;
1169         vfsops_t        *vfsops;
1170         struct vfs      *vfsp;
1171         struct vnode    *bvp;
1172         dev_t           bdev = 0;
1173         mntopts_t       mnt_mntopts;
1174         int             error = 0;
1175         int             copyout_error = 0;
1176         int             ovflags;
1177         char            *opts = uap->optptr;
1178         char            *inargs = opts;
1179         int             optlen = uap->optlen;
1180         int             remount;
1181         int             rdonly;
1182         int             nbmand = 0;
1183         int             delmip = 0;
```

```
1184        int             addmip = 0;
1185        int             splice = ((uap->flags & MS_NOSPLICE) == 0);
1186        int             fromspace = (uap->flags & MS_SYSSPACE) ?
1187            UIO_SYSSPACE : UIO_USERSPACE;
1188        char            *resource = NULL, *mountpt = NULL;
1189        refstr_t        *oldresource, *oldmntpt;
1190        struct pathname pn, rpn;
1191        vsk_anchor_t    *vskap;
1192        char fstname[FSTYPSZ];
1193        zone_t          *zone;

1195        /*
1196         * The v_flag value for the mount point vp is permanently set
1197         * to VVFSLOCK so that no one bypasses the vn_vfs*locks routine
1198         * for mount point locking.
1199         */
1200        mutex_enter(&vp->v_lock);
1201        vp->v_flag |= VVFSLOCK;
1202        mutex_exit(&vp->v_lock);

1204        mnt_mntopts.mo_count = 0;
1205        /*
1206         * Find the ops vector to use to invoke the file system-specific mount
1207         * method.  If the fsname argument is non-NULL, use it directly.
1208         * Otherwise, dig the file system type information out of the mount
1209         * arguments.
1210         *
1211         * A side effect is to hold the vfssw entry.
1212         *
1213         * Mount arguments can be specified in several ways, which are
1214         * distinguished by flag bit settings.  The preferred way is to set
1215         * MS_OPTIONSTR, indicating an 8 argument mount with the file system
1216         * type supplied as a character string and the last two arguments
1217         * being a pointer to a character buffer and the size of the buffer.
1218         * On entry, the buffer holds a null terminated list of options; on
1219         * return, the string is the list of options the file system
1220         * recognized. If MS_DATA is set arguments five and six point to a
1221         * block of binary data which the file system interprets.
1222         * A further wrinkle is that some callers don't set MS_FSS and MS_DATA
1223         * consistently with these conventions.  To handle them, we check to
1224         * see whether the pointer to the file system name has a numeric value
1225         * less than 256.  If so, we treat it as an index.
1226         */
1227        if (fsname != NULL) {
1228                if ((vswp = vfs_getvfssw(fsname)) == NULL) {
1229                        return (EINVAL);
1230                }
1231        } else if (uap->flags & (MS_OPTIONSTR | MS_DATA | MS_FSS)) {
1232                size_t n;
1233                uint_t fstype;

1235                fsname = fstname;

1237                if ((fstype = (uintptr_t)uap->fstype) < 256) {
1238                        RLOCK_VFSSW();
1239                        if (fstype == 0 || fstype >= nfstype ||
1240                            !ALLOCATED_VFSSW(&vfssw[fstype])) {
1241                                RUNLOCK_VFSSW();
1242                                return (EINVAL);
1243                        }
1244                        (void) strcpy(fsname, vfssw[fstype].vsw_name);
1245                        RUNLOCK_VFSSW();
1246                        if ((vswp = vfs_getvfssw(fsname)) == NULL)
1247                                return (EINVAL);
1248                } else {
1249                        /*
```



```
1250                         * Handle either kernel or user address space.
1251                         */
1252                        if (uap->flags & MS_SYSSPACE) {
1253                                error = copystr(uap->fstype, fsname,
1254                                    FSTYPSZ, &n);
1255                        } else {
1256                                error = copyinstr(uap->fstype, fsname,
1257                                    FSTYPSZ, &n);
1258                        }
1259                        if (error) {
1260                                if (error == ENAMETOOLONG)
1261                                        return (EINVAL);
1262                                return (error);
1263                        }
1264                        if ((vswp = vfs_getvfssw(fsname)) == NULL)
1265                                return (EINVAL);
1266                }
1267        } else {
1268                if ((vswp = vfs_getvfsswbyvfsops(vfs_getops(rootvfs))) == NULL)
1269                        return (EINVAL);
1270                fsname = vswp->vsw_name;
1271        }
1272        if (!VFS_INSTALLED(vswp))
1273                return (EINVAL);

1275        if ((error = secpolicy_fs_allowed_mount(fsname)) != 0)  {
1276                vfs_unrefvfssw(vswp);
1277                return (error);
1278        }

1280        vfsops = &vswp->vsw_vfsops;

1282        vfs_copyopttbl(&vswp->vsw_optproto, &mnt_mntopts);
1283        /*
1284         * Fetch mount options and parse them for generic vfs options
1285         */
1286        if (uap->flags & MS_OPTIONSTR) {
1287                /*
1288                 * Limit the buffer size
1289                 */
1290                if (optlen < 0 || optlen > MAX_MNTOPT_STR) {
1291                        error = EINVAL;
1292                        goto errout;
1293                }
1294                if ((uap->flags & MS_SYSSPACE) == 0) {
1295                        inargs = kmem_alloc(MAX_MNTOPT_STR, KM_SLEEP);
1296                        inargs[0] = '\0';
1297                        if (optlen) {
1298                                error = copyinstr(opts, inargs, (size_t)optlen,
1299                                    NULL);
1300                                if (error) {
1301                                        goto errout;
1302                                }
1303                        }
1304                }
1305                vfs_parsemntopts(&mnt_mntopts, inargs, 0);
1306        }
1307        /*
1308         * Flag bits override the options string.
1309         */
1310        if (uap->flags & MS_REMOUNT)
1311                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_REMOUNT, NULL, 0, 0);
1312        if (uap->flags & MS_RDONLY)
1313                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_RO, NULL, 0, 0);
1314        if (uap->flags & MS_NOSUID)
1315                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL, 0, 0);
```

```
1317         /*
1318          * Check if this is a remount; must be set in the option string and
1319          * the file system must support a remount option.
1320          */
1321         if (remount = vfs_optionisset_nolock(&mnt_mntopts,
1322             MNTOPT_REMOUNT, NULL)) {
1323                 if (!(vswp->vsw_flag & VSW_CANREMOUNT)) {
1324                         error = ENOTSUP;
1325                         goto errout;
1326                 }
1327                 uap->flags |= MS_REMOUNT;
1328         }
1329
1330         /*
1331          * uap->flags and vfs_optionisset() should agree.
1332          */
1333         if (rdonly = vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_RO, NULL)) {
1334                 uap->flags |= MS_RDONLY;
1335         }
1336         if (vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL)) {
1337                 uap->flags |= MS_NOSUID;
1338         }
1339         nbmand = vfs_optionisset_nolock(&mnt_mntopts, MNTOPT_NBMAND, NULL);
1340         ASSERT(splice || !remount);
1341         /*
1342          * If we are splicing the fs into the namespace,
1343          * perform mount point checks.
1344          *
1345          * We want to resolve the path for the mount point to eliminate
1346          * '.' and ".." and symlinks in mount points; we can't do the
1347          * same for the resource string, since it would turn
1348          * "/dev/dsk/c0t0d0s0" into "/devices/pci@...".  We need to do
1349          * this before grabbing vn_vfswlock(), because otherwise we
1350          * would deadlock with lookuppn().
1351          */
1352         if (splice) {
1353                 ASSERT(vp->v_count > 0);
1354
1355                 /*
1356                  * Pick up mount point and device from appropriate space.
1357                  */
1358                 if (pn_get(uap->spec, fromspace, &pn) == 0) {
1359                         resource = kmem_alloc(pn.pn_pathlen + 1,
1360                             KM_SLEEP);
1361                         (void) strcpy(resource, pn.pn_path);
1362                         pn_free(&pn);
1363                 }
1364                 /*
1365                  * Do a lookupname prior to taking the
1366                  * writelock. Mark this as completed if
1367                  * successful for later cleanup and addition to
1368                  * the mount in progress table.
1369                  */
1370                 if ((uap->flags & MS_GLOBAL) == 0 &&
1371                     lookupname(uap->spec, fromspace,
1372                     FOLLOW, NULL, &bvp) == 0) {
1373                         addmip = 1;
1374                 }
1375
1376                 if ((error = pn_get(uap->dir, fromspace, &pn)) == 0) {
1377                         pathname_t *pnp;
1378
1379                         if (*pn.pn_path != '/') {
1380                                 error = EINVAL;
1381                                 pn_free(&pn);
```



```
1382                                 goto errout;
1383                         }
1384                         pn_alloc(&rpn);
1385                         /*
1386                          * Kludge to prevent autofs from deadlocking with
1387                          * itself when it calls domount().
1388                          *
1389                          * If autofs is calling, it is because it is doing
1390                          * (autofs) mounts in the process of an NFS mount.  A
1391                          * lookuppn() here would cause us to block waiting for
1392                          * said NFS mount to complete, which can't since this
1393                          * is the thread that was supposed to doing it.
1394                          */
1395                         if (fromspace == UIO_USERSPACE) {
1396                                 if ((error = lookuppn(&pn, &rpn, FOLLOW, NULL,
1397                                     NULL)) == 0) {
1398                                         pnp = &rpn;
1399                                 } else {
1400                                         /*
1401                                          * The file disappeared or otherwise
1402                                          * became inaccessible since we opened
1403                                          * it; might as well fail the mount
1404                                          * since the mount point is no longer
1405                                          * accessible.
1406                                          */
1407                                         pn_free(&rpn);
1408                                         pn_free(&pn);
1409                                         goto errout;
1410                                 }
1411                         } else {
1412                                 pnp = &pn;
1413                         }
1414                         mountpt = kmem_alloc(pnp->pn_pathlen + 1, KM_SLEEP);
1415                         (void) strcpy(mountpt, pnp->pn_path);
1416
1417                         /*
1418                          * If the addition of the zone's rootpath
1419                          * would push us over a total path length
1420                          * of MAXPATHLEN, we fail the mount with
1421                          * ENAMETOOLONG, which is what we would have
1422                          * gotten if we were trying to perform the same
1423                          * mount in the global zone.
1424                          *
1425                          * strlen() doesn't count the trailing
1426                          * '\0', but zone_rootpathlen counts both a
1427                          * trailing '/' and the terminating '\0'.
1428                          */
1429                         if ((curproc->p_zone->zone_rootpathlen - 1 +
1430                             strlen(mountpt)) > MAXPATHLEN ||
1431                             (resource != NULL &&
1432                             (curproc->p_zone->zone_rootpathlen - 1 +
1433                             strlen(resource)) > MAXPATHLEN)) {
1434                                 error = ENAMETOOLONG;
1435                         }
1436
1437                         pn_free(&rpn);
1438                         pn_free(&pn);
1439                 }
1440
1441                 if (error)
1442                         goto errout;
1443
1444                 /*
1445                  * Prevent path name resolution from proceeding past
1446                  * the mount point.
1447                  */
```

```
1448                if (vn_vfswlock(vp) != 0) {
1449                        error = EBUSY;
1450                        goto errout;
1451                }

1453                /*
1454                 * Verify that it's legitimate to establish a mount on
1455                 * the prospective mount point.
1456                 */
1457                if (vn_mountedvfs(vp) != NULL) {
1458                        /*
1459                         * The mount point lock was obtained after some
1460                         * other thread raced through and established a mount.
1461                         */
1462                        vn_vfsunlock(vp);
1463                        error = EBUSY;
1464                        goto errout;
1465                }
1466                if (vp->v_flag & VNOMOUNT) {
1467                        vn_vfsunlock(vp);
1468                        error = EINVAL;
1469                        goto errout;
1470                }
1471        }
1472        if ((uap->flags & (MS_DATA | MS_OPTIONSTR)) == 0) {
1473                uap->dataptr = NULL;
1474                uap->datalen = 0;
1475        }

1477        /*
1478         * If this is a remount, we don't want to create a new VFS.
1479         * Instead, we pass the existing one with a remount flag.
1480         */
1481        if (remount) {
1482                /*
1483                 * Confirm that the mount point is the root vnode of the
1484                 * file system that is being remounted.
1485                 * This can happen if the user specifies a different
1486                 * mount point directory pathname in the (re)mount command.
1487                 *
1488                 * Code below can only be reached if splice is true, so it's
1489                 * safe to do vn_vfsunlock() here.
1490                 */
1491                if ((vp->v_flag & VROOT) == 0) {
1492                        vn_vfsunlock(vp);
1493                        error = ENOENT;
1494                        goto errout;
1495                }
1496                /*
1497                 * Disallow making file systems read-only unless file system
1498                 * explicitly allows it in its vfssw.  Ignore other flags.
1499                 */
1500                if (rdonly && vn_is_readonly(vp) == 0 &&
1501                    (vswp->vsw_flag & VSW_CANRWRO) == 0) {
1502                        vn_vfsunlock(vp);
1503                        error = EINVAL;
1504                        goto errout;
1505                }
1506                /*
1507                 * Disallow changing the NBMAND disposition of the file
1508                 * system on remounts.
1509                 */
1510                if ((nbmand && ((vp->v_vfsp->vfs_flag & VFS_NBMAND) == 0)) ||
1511                    (!nbmand && (vp->v_vfsp->vfs_flag & VFS_NBMAND))) {
1512                        vn_vfsunlock(vp);
1513                        error = EINVAL;
```



```
1514                        goto errout;
1515                }
1516                vfsp = vp->v_vfsp;
1517                ovflags = vfsp->vfs_flag;
1518                vfsp->vfs_flag |= VFS_REMOUNT;
1519                vfsp->vfs_flag &= ~VFS_RDONLY;
1520        } else {
1521                vfsp = vfs_alloc(KM_SLEEP);
1522                VFS_INIT(vfsp, vfsops, NULL);
1523        }

1525        VFS_HOLD(vfsp);

1527        if ((error = lofi_add(fsname, vfsp, &mnt_mntopts, uap)) != 0) {
1528                if (!remount) {
1529                        if (splice)
1530                                vn_vfsunlock(vp);
1531                        vfs_free(vfsp);
1532                } else {
1533                        vn_vfsunlock(vp);
1534                        VFS_RELE(vfsp);
1535                }
1536                goto errout;
1537        }

1539        /*
1540         * PRIV_SYS_MOUNT doesn't mean you can become root.
1541         */
1542        if (vfsp->vfs_lofi_minor != 0) {
1543                uap->flags |= MS_NOSUID;
1544                vfs_setmntopt_nolock(&mnt_mntopts, MNTOPT_NOSUID, NULL, 0, 0);
1545        }

1547        /*
1548         * The vfs_reflock is not used anymore the code below explicitly
1549         * holds it preventing others accesing it directly.
1550         */
1551        if ((sema_tryp(&vfsp->vfs_reflock) == 0) &&
1552            !(vfsp->vfs_flag & VFS_REMOUNT))
1553                cmn_err(CE_WARN,
1554                    "mount type %s couldn't get vfs_reflock", vswp->vsw_name);

1556        /*
1557         * Lock the vfs. If this is a remount we want to avoid spurious umount
1558         * failures that happen as a side-effect of fsflush() and other mount
1559         * and unmount operations that might be going on simultaneously and
1560         * may have locked the vfs currently. To not return EBUSY immediately
1561         * here we use vfs_lock_wait() instead vfs_lock() for the remount case.
1562         */
1563        if (!remount) {
1564                if (error = vfs_lock(vfsp)) {
1565                        vfsp->vfs_flag = ovflags;

1567                        lofi_remove(vfsp);

1569                        if (splice)
1570                                vn_vfsunlock(vp);
1571                        vfs_free(vfsp);
1572                        goto errout;
1573                }
1574        } else {
1575                vfs_lock_wait(vfsp);
1576        }

1578        /*
1579         * Add device to mount in progress table, global mounts require special
```

```
1580          * handling. It is possible that we have already done the lookupname
1581          * on a spliced, non-global fs. If so, we don't want to do it again
1582          * since we cannot do a lookupname after taking the
1583          * wlock above. This case is for a non-spliced, non-global filesystem.
1584          */
1585         if (!addmip) {
1586                 if ((uap->flags & MS_GLOBAL) == 0 &&
1587                     lookupname(uap->spec, fromspace, FOLLOW, NULL, &bvp) == 0) {
1588                         addmip = 1;
1589                 }
1590         }
1591
1592         if (addmip) {
1593                 vnode_t *lvp = NULL;
1594
1595                 error = vfs_get_lofi(vfsp, &lvp);
1596                 if (error > 0) {
1597                         lofi_remove(vfsp);
1598
1599                         if (splice)
1600                                 vn_vfsunlock(vp);
1601                         vfs_unlock(vfsp);
1602
1603                         if (remount) {
1604                                 VFS_RELE(vfsp);
1605                         } else {
1606                                 vfs_free(vfsp);
1607                         }
1608
1609                         goto errout;
1610                 } else if (error == -1) {
1611                         bdev = bvp->v_rdev;
1612                         VN_RELE(bvp);
1613                 } else {
1614                         bdev = lvp->v_rdev;
1615                         VN_RELE(lvp);
1616                         VN_RELE(bvp);
1617                 }
1618
1619                 vfs_addmip(bdev, vfsp);
1620                 addmip = 0;
1621                 delmip = 1;
1622         }
1623         /*
1624          * Invalidate cached entry for the mount point.
1625          */
1626         if (splice)
1627                 dnlc_purge_vp(vp);
1628
1629         /*
1630          * If have an option string but the filesystem doesn't supply a
1631          * prototype options table, create a table with the global
1632          * options and sufficient room to accept all the options in the
1633          * string.  Then parse the passed in option string
1634          * accepting all the options in the string.  This gives us an
1635          * option table with all the proper cancel properties for the
1636          * global options.
1637          *
1638          * Filesystems that supply a prototype options table are handled
1639          * earlier in this function.
1640          */
1641         if (uap->flags & MS_OPTIONSTR) {
1642                 if (!(vswp->vsw_flag & VSW_HASPROTO)) {
1643                         mntopts_t tmp_mntopts;
1644
1645                         tmp_mntopts.mo_count = 0;
```



```
1646                         vfs_createopttbl_extend(&tmp_mntopts, inargs,
1647                             &mnt_mntopts);
1648                         vfs_parsemntopts(&tmp_mntopts, inargs, 1);
1649                         vfs_swapopttbl_nolock(&mnt_mntopts, &tmp_mntopts);
1650                         vfs_freeopttbl(&tmp_mntopts);
1651                 }
1652         }
1653
1654         /*
1655          * Serialize with zone state transitions.
1656          * See vfs_list_add; zone mounted into is:
1657          *      zone_find_by_path(refstr_value(vfsp->vfs_mntpt))
1658          * not the zone doing the mount (curproc->p_zone), but if we're already
1659          * inside a NGZ, then we know what zone we are.
1660          */
1661         if (INGLOBALZONE(curproc)) {
1662                 zone = zone_find_by_path(mountpt);
1663                 ASSERT(zone != NULL);
1664         } else {
1665                 zone = curproc->p_zone;
1666                 /*
1667                  * zone_find_by_path does a hold, so do one here too so that
1668                  * we can do a zone_rele after mount_completed.
1669                  */
1670                 zone_hold(zone);
1671         }
1672         mount_in_progress(zone);
1673         /*
1674          * Instantiate (or reinstantiate) the file system.  If appropriate,
1675          * splice it into the file system name space.
1676          *
1677          * We want VFS_MOUNT() to be able to override the vfs_resource
1678          * string if necessary (ie, mntfs), and also for a remount to
1679          * change the same (necessary when remounting '/' during boot).
1680          * So we set up vfs_mntpt and vfs_resource to what we think they
1681          * should be, then hand off control to VFS_MOUNT() which can
1682          * override this.
1683          *
1684          * For safety's sake, when changing vfs_resource or vfs_mntpt of
1685          * a vfs which is on the vfs list (i.e. during a remount), we must
1686          * never set those fields to NULL. Several bits of code make
1687          * assumptions that the fields are always valid.
1688          */
1689         vfs_swapopttbl(&mnt_mntopts, &vfsp->vfs_mntopts);
1690         if (remount) {
1691                 if ((oldresource = vfsp->vfs_resource) != NULL)
1692                         refstr_hold(oldresource);
1693                 if ((oldmntpt = vfsp->vfs_mntpt) != NULL)
1694                         refstr_hold(oldmntpt);
1695         }
1696         vfs_setresource(vfsp, resource, 0);
1697         vfs_setmntpoint(vfsp, mountpt, 0);
1698
1699         /*
1700          * going to mount on this vnode, so notify.
1701          */
1702         vnevent_mountedover(vp, NULL);
1703         error = VFS_MOUNT(vfsp, vp, uap, credp);
1704
1705         if (uap->flags & MS_RDONLY)
1706                 vfs_setmntopt(vfsp, MNTOPT_RO, NULL, 0);
1707         if (uap->flags & MS_NOSUID)
1708                 vfs_setmntopt(vfsp, MNTOPT_NOSUID, NULL, 0);
1709         if (uap->flags & MS_GLOBAL)
1710                 vfs_setmntopt(vfsp, MNTOPT_GLOBAL, NULL, 0);
```

```
1712         if (error) {
1713                 lofi_remove(vfsp);

1715                 if (remount) {
1716                         /* put back pre-remount options */
1717                         vfs_swapopttbl(&mnt_mntopts, &vfsp->vfs_mntopts);
1718                         vfs_setmntpoint(vfsp, refstr_value(oldmntpt),
1719                             VFSSP_VERBATIM);
1720                         if (oldmntpt)
1721                                 refstr_rele(oldmntpt);
1722                         vfs_setresource(vfsp, refstr_value(oldresource),
1723                             VFSSP_VERBATIM);
1724                         if (oldresource)
1725                                 refstr_rele(oldresource);
1726                         vfsp->vfs_flag = ovflags;
1727                         vfs_unlock(vfsp);
1728                         VFS_RELE(vfsp);
1729                 } else {
1730                         vfs_unlock(vfsp);
1731                         vfs_freemnttab(vfsp);
1732                         vfs_free(vfsp);
1733                 }
1734         } else {
1735                 /*
1736                  * Set the mount time to now
1737                  */
1738                 vfsp->vfs_mtime = ddi_get_time();
1739                 if (remount) {
1740                         vfsp->vfs_flag &= ~VFS_REMOUNT;
1741                         if (oldresource)
1742                                 refstr_rele(oldresource);
1743                         if (oldmntpt)
1744                                 refstr_rele(oldmntpt);
1745                 } else if (splice) {
1746                         /*
1747                          * Link vfsp into the name space at the mount
1748                          * point. Vfs_add() is responsible for
1749                          * holding the mount point which will be
1750                          * released when vfs_remove() is called.
1751                          */
1752                         vfs_add(vp, vfsp, uap->flags);
1753                 } else {
1754                         /*
1755                          * Hold the reference to file system which is
1756                          * not linked into the name space.
1757                          */
1758                         vfsp->vfs_zone = NULL;
1759                         VFS_HOLD(vfsp);
1760                         vfsp->vfs_vnodecovered = NULL;
1761                 }
1762                 /*
1763                  * Set flags for global options encountered
1764                  */
1765                 if (vfs_optionisset(vfsp, MNTOPT_RO, NULL))
1766                         vfsp->vfs_flag |= VFS_RDONLY;
1767                 else
1768                         vfsp->vfs_flag &= ~VFS_RDONLY;
1769                 if (vfs_optionisset(vfsp, MNTOPT_NOSUID, NULL)) {
1770                         vfsp->vfs_flag |= (VFS_NOSETUID|VFS_NODEVICES);
1771                 } else {
1772                         if (vfs_optionisset(vfsp, MNTOPT_NODEVICES, NULL))
1773                                 vfsp->vfs_flag |= VFS_NODEVICES;
1774                         else
1775                                 vfsp->vfs_flag &= ~VFS_NODEVICES;
1776                         if (vfs_optionisset(vfsp, MNTOPT_NOSETUID, NULL))
1777                                 vfsp->vfs_flag |= VFS_NOSETUID;
```



```
1778                         else
1779                                 vfsp->vfs_flag &= ~VFS_NOSETUID;
1780                 }
1781                 if (vfs_optionisset(vfsp, MNTOPT_NBMAND, NULL))
1782                         vfsp->vfs_flag |= VFS_NBMAND;
1783                 else
1784                         vfsp->vfs_flag &= ~VFS_NBMAND;

1786                 if (vfs_optionisset(vfsp, MNTOPT_XATTR, NULL))
1787                         vfsp->vfs_flag |= VFS_XATTR;
1788                 else
1789                         vfsp->vfs_flag &= ~VFS_XATTR;

1791                 if (vfs_optionisset(vfsp, MNTOPT_NOEXEC, NULL))
1792                         vfsp->vfs_flag |= VFS_NOEXEC;
1793                 else
1794                         vfsp->vfs_flag &= ~VFS_NOEXEC;

1796                 /*
1797                  * Now construct the output option string of options
1798                  * we recognized.
1799                  */
1800                 if (uap->flags & MS_OPTIONSTR) {
1801                         vfs_list_read_lock();
1802                         copyout_error = vfs_buildoptionstr(
1803                             &vfsp->vfs_mntopts, inargs, optlen);
1804                         vfs_list_unlock();
1805                         if (copyout_error == 0 &&
1806                             (uap->flags & MS_SYSSPACE) == 0) {
1807                                 copyout_error = copyoutstr(inargs, opts,
1808                                     optlen, NULL);
1809                         }
1810                 }

1812                 /*
1813                  * If this isn't a remount, set up the vopstats before
1814                  * anyone can touch this. We only allow spliced file
1815                  * systems (file systems which are in the namespace) to
1816                  * have the VFS_STATS flag set.
1817                  * NOTE: PxFS mounts the underlying file system with
1818                  * MS_NOSPLICE set and copies those vfs_flags to its private
1819                  * vfs structure. As a result, PxFS should never have
1820                  * the VFS_STATS flag or else we might access the vfs
1821                  * statistics-related fields prior to them being
1822                  * properly initialized.
1823                  */
1824                 if (!remount && (vswp->vsw_flag & VSW_STATS) && splice) {
1825                         initialize_vopstats(&vfsp->vfs_vopstats);
1826                         /*
1827                          * We need to set vfs_vskap to NULL because there's
1828                          * a chance it won't be set below.  This is checked
1829                          * in teardown_vopstats() so we can't have garbage.
1830                          */
1831                         vfsp->vfs_vskap = NULL;
1832                         vfsp->vfs_flag |= VFS_STATS;
1833                         vfsp->vfs_fstypevsp = get_fstype_vopstats(vfsp, vswp);
1834                 }

1836                 if (vswp->vsw_flag & VSW_XID)
1837                         vfsp->vfs_flag |= VFS_XID;

1839                 vfs_unlock(vfsp);
1840         }
1841         mount_completed(zone);
1842         zone_rele(zone);
1843         if (splice)
```

```
1844                 vn_vfsunlock(vp);

1846         if ((error == 0) && (copyout_error == 0)) {
1847                 if (!remount) {
1848                         /*
1849                          * Don't call get_vskstat_anchor() while holding
1850                          * locks since it allocates memory and calls
1851                          * VFS_STATVFS().  For NFS, the latter can generate
1852                          * an over-the-wire call.
1853                          */
1854                         vskap = get_vskstat_anchor(vfsp);
1855                         /* Only take the lock if we have something to do */
1856                         if (vskap != NULL) {
1857                                 vfs_lock_wait(vfsp);
1858                                 if (vfsp->vfs_flag & VFS_STATS) {
1859                                         vfsp->vfs_vskap = vskap;
1860                                 }
1861                                 vfs_unlock(vfsp);
1862                         }
1863                 }
1864                 /* Return vfsp to caller. */
1865                 *vfspp = vfsp;
1866         }
1867 errout:
1868         vfs_freeopttbl(&mnt_mntopts);
1869         if (resource != NULL)
1870                 kmem_free(resource, strlen(resource) + 1);
1871         if (mountpt != NULL)
1872                 kmem_free(mountpt, strlen(mountpt) + 1);
1873         /*
1874          * It is possible we errored prior to adding to mount in progress
1875          * table. Must free vnode we acquired with successful lookupname.
1876          */
1877         if (addmip)
1878                 VN_RELE(bvp);
1879         if (delmip)
1880                 vfs_delmip(vfsp);
1881         ASSERT(vswp != NULL);
1882         vfs_unrefvfssw(vswp);
1883         if (inargs != opts)
1884                 kmem_free(inargs, MAX_MNTOPT_STR);
1885         if (copyout_error) {
1886                 lofi_remove(vfsp);
1887                 VFS_RELE(vfsp);
1888                 error = copyout_error;
1889         }
1890         return (error);
1891 }

1893 static void
1894 vfs_setpath(
1895     struct vfs *vfsp,           /* vfs being updated */
1896     refstr_t **refp,            /* Ref-count string to contain the new path */
1897     const char *newpath,        /* Path to add to refp (above) */
1898     uint32_t flag)              /* flag */
1899 {
1900         size_t len;
1901         refstr_t *ref;
1902         zone_t *zone = curproc->p_zone;
1903         char *sp;
1904         int have_list_lock = 0;

1906         ASSERT(!VFS_ON_LIST(vfsp) || vfs_lock_held(vfsp));

1908         /*
1909          * New path must be less than MAXPATHLEN because mntfs
```



```
1910          * will only display up to MAXPATHLEN bytes. This is currently
1911          * safe, because domount() uses pn_get(), and other callers
1912          * similarly cap the size to fewer than MAXPATHLEN bytes.
1913          */

1915         ASSERT(strlen(newpath) < MAXPATHLEN);

1917         /* mntfs requires consistency while vfs list lock is held */

1919         if (VFS_ON_LIST(vfsp)) {
1920                 have_list_lock = 1;
1921                 vfs_list_lock();
1922         }

1924         if (*refp != NULL)
1925                 refstr_rele(*refp);

1927         /*
1928          * If we are in a non-global zone then we prefix the supplied path,
1929          * newpath, with the zone's root path, with two exceptions. The first
1930          * is where we have been explicitly directed to avoid doing so; this
1931          * will be the case following a failed remount, where the path supplied
1932          * will be a saved version which must now be restored. The second
1933          * exception is where newpath is not a pathname but a descriptive name,
1934          * e.g. "procfs".
1935          */
1936         if (zone == global_zone || (flag & VFSSP_VERBATIM) || *newpath != '/') {
1937                 ref = refstr_alloc(newpath);
1938                 goto out;
1939         }

1941         /*
1942          * Truncate the trailing '/' in the zoneroot, and merge
1943          * in the zone's rootpath with the "newpath" (resource
1944          * or mountpoint) passed in.
1945          *
1946          * The size of the required buffer is thus the size of
1947          * the buffer required for the passed-in newpath
1948          * (strlen(newpath) + 1), plus the size of the buffer
1949          * required to hold zone_rootpath (zone_rootpathlen)
1950          * minus one for one of the now-superfluous NUL
1951          * terminations, minus one for the trailing '/'.
1952          *
1953          * That gives us:
1954          *
1955          * (strlen(newpath) + 1) + zone_rootpathlen - 1 - 1
1956          *
1957          * Which is what we have below.
1958          */

1960         len = strlen(newpath) + zone->zone_rootpathlen - 1;
1961         sp = kmem_alloc(len, KM_SLEEP);

1963         /*
1964          * Copy everything including the trailing slash, which
1965          * we then overwrite with the NUL character.
1966          */

1968         (void) strcpy(sp, zone->zone_rootpath);
1969         sp[zone->zone_rootpathlen - 2] = '\0';
1970         (void) strcat(sp, newpath);

1972         ref = refstr_alloc(sp);
1973         kmem_free(sp, len);
1974 out:
1975         *refp = ref;
```

```
1977         if (have_list_lock) {
1978                 vfs_mnttab_modtimeupd();
1979                 vfs_list_unlock();
1980         }
1981 }

1983 /*
1984  * Record a mounted resource name in a vfs structure.
1985  * If vfsp is already mounted, caller must hold the vfs lock.
1986  */
1987 void
1988 vfs_setresource(struct vfs *vfsp, const char *resource, uint32_t flag)
1989 {
1990         if (resource == NULL || resource[0] == '\0')
1991                 resource = VFS_NORESOURCE;
1992         vfs_setpath(vfsp, &vfsp->vfs_resource, resource, flag);
1993 }

1995 /*
1996  * Record a mount point name in a vfs structure.
1997  * If vfsp is already mounted, caller must hold the vfs lock.
1998  */
1999 void
2000 vfs_setmntpoint(struct vfs *vfsp, const char *mntpt, uint32_t flag)
2001 {
2002         if (mntpt == NULL || mntpt[0] == '\0')
2003                 mntpt = VFS_NOMNTPT;
2004         vfs_setpath(vfsp, &vfsp->vfs_mntpt, mntpt, flag);
2005 }

2007 /* Returns the vfs_resource. Caller must call refstr_rele() when finished. */

2009 refstr_t *
2010 vfs_getresource(const struct vfs *vfsp)
2011 {
2012         refstr_t *resource;

2014         vfs_list_read_lock();
2015         resource = vfsp->vfs_resource;
2016         refstr_hold(resource);
2017         vfs_list_unlock();

2019         return (resource);
2020 }

2022 /* Returns the vfs_mntpt. Caller must call refstr_rele() when finished. */

2024 refstr_t *
2025 vfs_getmntpoint(const struct vfs *vfsp)
2026 {
2027         refstr_t *mntpt;

2029         vfs_list_read_lock();
2030         mntpt = vfsp->vfs_mntpt;
2031         refstr_hold(mntpt);
2032         vfs_list_unlock();

2034         return (mntpt);
2035 }

2037 /*
2038  * Create an empty options table with enough empty slots to hold all
2039  * The options in the options string passed as an argument.
2040  * Potentially prepend another options table.
2041  *
```



```
2042  * Note: caller is responsible for locking the vfs list, if needed,
2043  *       to protect mops.
2044  */
2045 static void
2046 vfs_createopttbl_extend(mntopts_t *mops, const char *opts,
2047     const mntopts_t *mtmpl)
2048 {
2049         const char *s = opts;
2050         uint_t count;

2052         if (opts == NULL || *opts == '\0') {
2053                 count = 0;
2054         } else {
2055                 count = 1;

2057                 /*
2058                  * Count number of options in the string
2059                  */
2060                 for (s = strchr(s, ','); s != NULL; s = strchr(s, ',')) {
2061                         count++;
2062                         s++;
2063                 }
2064         }
2065         vfs_copyopttbl_extend(mtmpl, mops, count);
2066 }

2068 /*
2069  * Create an empty options table with enough empty slots to hold all
2070  * The options in the options string passed as an argument.
2071  *
2072  * This function is *not* for general use by filesystems.
2073  *
2074  * Note: caller is responsible for locking the vfs list, if needed,
2075  *       to protect mops.
2076  */
2077 void
2078 vfs_createopttbl(mntopts_t *mops, const char *opts)
2079 {
2080         vfs_createopttbl_extend(mops, opts, NULL);
2081 }


2084 /*
2085  * Swap two mount options tables
2086  */
2087 static void
2088 vfs_swapopttbl_nolock(mntopts_t *optbl1, mntopts_t *optbl2)
2089 {
2090         uint_t tmpcnt;
2091         mntopt_t *tmplist;

2093         tmpcnt = optbl2->mo_count;
2094         tmplist = optbl2->mo_list;
2095         optbl2->mo_count = optbl1->mo_count;
2096         optbl2->mo_list = optbl1->mo_list;
2097         optbl1->mo_count = tmpcnt;
2098         optbl1->mo_list = tmplist;
2099 }

2101 static void
2102 vfs_swapopttbl(mntopts_t *optbl1, mntopts_t *optbl2)
2103 {
2104         vfs_list_lock();
2105         vfs_swapopttbl_nolock(optbl1, optbl2);
2106         vfs_mnttab_modtimeupd();
2107         vfs_list_unlock();
```

```
2108 }
2110 static char **
2111 vfs_copycancelopt_extend(char **const moc, int extend)
2112 {
2113         int i = 0;
2114         int j;
2115         char **result;
2117         if (moc != NULL) {
2118                 for (; moc[i] != NULL; i++)
2119                         /* count number of options to cancel */;
2120         }
2122         if (i + extend == 0)
2123                 return (NULL);
2125         result = kmem_alloc((i + extend + 1) * sizeof (char *), KM_SLEEP);
2127         for (j = 0; j < i; j++) {
2128                 result[j] = kmem_alloc(strlen(moc[j]) + 1, KM_SLEEP);
2129                 (void) strcpy(result[j], moc[j]);
2130         }
2131         for (; j <= i + extend; j++)
2132                 result[j] = NULL;
2134         return (result);
2135 }
2137 static void
2138 vfs_copyopt(const mntopt_t *s, mntopt_t *d)
2139 {
2140         char *sp, *dp;
2142         d->mo_flags = s->mo_flags;
2143         d->mo_data = s->mo_data;
2144         sp = s->mo_name;
2145         if (sp != NULL) {
2146                 dp = kmem_alloc(strlen(sp) + 1, KM_SLEEP);
2147                 (void) strcpy(dp, sp);
2148                 d->mo_name = dp;
2149         } else {
2150                 d->mo_name = NULL; /* should never happen */
2151         }
2153         d->mo_cancel = vfs_copycancelopt_extend(s->mo_cancel, 0);
2155         sp = s->mo_arg;
2156         if (sp != NULL) {
2157                 dp = kmem_alloc(strlen(sp) + 1, KM_SLEEP);
2158                 (void) strcpy(dp, sp);
2159                 d->mo_arg = dp;
2160         } else {
2161                 d->mo_arg = NULL;
2162         }
2163 }
2165 /*
2166  * Copy a mount options table, possibly allocating some spare
2167  * slots at the end.  It is permissible to copy_extend the NULL table.
2168  */
2169 static void
2170 vfs_copyopttbl_extend(const mntopts_t *smo, mntopts_t *dmo, int extra)
2171 {
2172         uint_t i, count;
2173         mntopt_t *motbl;
```



```
2175         /*
2176          * Clear out any existing stuff in the options table being initialized
2177          */
2178         vfs_freeopttbl(dmo);
2179         count = (smo == NULL) ? 0 : smo->mo_count;
2180         if ((count + extra) == 0)       /* nothing to do */
2181                 return;
2182         dmo->mo_count = count + extra;
2183         motbl = kmem_zalloc((count + extra) * sizeof (mntopt_t), KM_SLEEP);
2184         dmo->mo_list = motbl;
2185         for (i = 0; i < count; i++) {
2186                 vfs_copyopt(&smo->mo_list[i], &motbl[i]);
2187         }
2188         for (i = count; i < count + extra; i++) {
2189                 motbl[i].mo_flags = MO_EMPTY;
2190         }
2191 }
2193 /*
2194  * Copy a mount options table.
2195  *
2196  * This function is *not* for general use by filesystems.
2197  *
2198  * Note: caller is responsible for locking the vfs list, if needed,
2199  *       to protect smo and dmo.
2200  */
2201 void
2202 vfs_copyopttbl(const mntopts_t *smo, mntopts_t *dmo)
2203 {
2204         vfs_copyopttbl_extend(smo, dmo, 0);
2205 }
2207 static char **
2208 vfs_mergecancelopts(const mntopt_t *mop1, const mntopt_t *mop2)
2209 {
2210         int c1 = 0;
2211         int c2 = 0;
2212         char **result;
2213         char **sp1, **sp2, **dp;
2215         /*
2216          * First we count both lists of cancel options.
2217          * If either is NULL or has no elements, we return a copy of
2218          * the other.
2219          */
2220         if (mop1->mo_cancel != NULL) {
2221                 for (; mop1->mo_cancel[c1] != NULL; c1++)
2222                         /* count cancel options in mop1 */;
2223         }
2225         if (c1 == 0)
2226                 return (vfs_copycancelopt_extend(mop2->mo_cancel, 0));
2228         if (mop2->mo_cancel != NULL) {
2229                 for (; mop2->mo_cancel[c2] != NULL; c2++)
2230                         /* count cancel options in mop2 */;
2231         }
2233         result = vfs_copycancelopt_extend(mop1->mo_cancel, c2);
2235         if (c2 == 0)
2236                 return (result);
2238         /*
2239          * When we get here, we've got two sets of cancel options;
```

```
2240          * we need to merge the two sets.  We know that the result
2241          * array has "c1+c2+1" entries and in the end we might shrink
2242          * it.
2243          * Result now has a copy of the c1 entries from mop1; we'll
2244          * now lookup all the entries of mop2 in mop1 and copy it if
2245          * it is unique.
2246          * This operation is O(n^2) but it's only called once per
2247          * filesystem per duplicate option.  This is a situation
2248          * which doesn't arise with the filesystems in ON and
2249          * n is generally 1.
2250          */

2252         dp = &result[c1];
2253         for (sp2 = mop2->mo_cancel; *sp2 != NULL; sp2++) {
2254                 for (sp1 = mop1->mo_cancel; *sp1 != NULL; sp1++) {
2255                         if (strcmp(*sp1, *sp2) == 0)
2256                                 break;
2257                 }
2258                 if (*sp1 == NULL) {
2259                         /*
2260                          * Option *sp2 not found in mop1, so copy it.
2261                          * The calls to vfs_copycancelopt_extend()
2262                          * guarantee that there's enough room.
2263                          */
2264                         *dp = kmem_alloc(strlen(*sp2) + 1, KM_SLEEP);
2265                         (void) strcpy(*dp++, *sp2);
2266                 }
2267         }
2268         if (dp != &result[c1+c2]) {
2269                 size_t bytes = (dp - result + 1) * sizeof (char *);
2270                 char **nres = kmem_alloc(bytes, KM_SLEEP);

2272                 bcopy(result, nres, bytes);
2273                 kmem_free(result, (c1 + c2 + 1) * sizeof (char *));
2274                 result = nres;
2275         }
2276         return (result);
2277 }

2279 /*
2280  * Merge two mount option tables (outer and inner) into one.  This is very
2281  * similar to "merging" global variables and automatic variables in C.
2282  *
2283  * This isn't (and doesn't have to be) fast.
2284  *
2285  * This function is *not* for general use by filesystems.
2286  *
2287  * Note: caller is responsible for locking the vfs list, if needed,
2288  *       to protect omo, imo & dmo.
2289  */
2290 void
2291 vfs_mergeopttbl(const mntopts_t *omo, const mntopts_t *imo, mntopts_t *dmo)
2292 {
2293         uint_t i, count;
2294         mntopt_t *mop, *motbl;
2295         uint_t freeidx;

2297         /*
2298          * First determine how much space we need to allocate.
2299          */
2300         count = omo->mo_count;
2301         for (i = 0; i < imo->mo_count; i++) {
2302                 if (imo->mo_list[i].mo_flags & MO_EMPTY)
2303                         continue;
2304                 if (vfs_hasopt(omo, imo->mo_list[i].mo_name) == NULL)
2305                         count++;
```



```
2306         }
2307         ASSERT(count >= omo->mo_count &&
2308             count <= omo->mo_count + imo->mo_count);
2309         motbl = kmem_alloc(count * sizeof (mntopt_t), KM_SLEEP);
2310         for (i = 0; i < omo->mo_count; i++)
2311                 vfs_copyopt(&omo->mo_list[i], &motbl[i]);
2312         freeidx = omo->mo_count;
2313         for (i = 0; i < imo->mo_count; i++) {
2314                 if (imo->mo_list[i].mo_flags & MO_EMPTY)
2315                         continue;
2316                 if ((mop = vfs_hasopt(omo, imo->mo_list[i].mo_name)) != NULL) {
2317                         char **newcanp;
2318                         uint_t index = mop - omo->mo_list;

2320                         newcanp = vfs_mergecancelopts(mop, &motbl[index]);

2322                         vfs_freeopt(&motbl[index]);
2323                         vfs_copyopt(&imo->mo_list[i], &motbl[index]);

2325                         vfs_freecancelopt(motbl[index].mo_cancel);
2326                         motbl[index].mo_cancel = newcanp;
2327                 } else {
2328                         /*
2329                          * If it's a new option, just copy it over to the first
2330                          * free location.
2331                          */
2332                         vfs_copyopt(&imo->mo_list[i], &motbl[freeidx++]);
2333                 }
2334         }
2335         dmo->mo_count = count;
2336         dmo->mo_list = motbl;
2337 }

2339 /*
2340  * Functions to set and clear mount options in a mount options table.
2341  */

2343 /*
2344  * Clear a mount option, if it exists.
2345  *
2346  * The update_mnttab arg indicates whether mops is part of a vfs that is on
2347  * the vfs list.
2348  */
2349 static void
2350 vfs_clearmntopt_nolock(mntopts_t *mops, const char *opt, int update_mnttab)
2351 {
2352         struct mntopt *mop;
2353         uint_t i, count;

2355         ASSERT(!update_mnttab || RW_WRITE_HELD(&vfslist));

2357         count = mops->mo_count;
2358         for (i = 0; i < count; i++) {
2359                 mop = &mops->mo_list[i];

2361                 if (mop->mo_flags & MO_EMPTY)
2362                         continue;
2363                 if (strcmp(opt, mop->mo_name))
2364                         continue;
2365                 mop->mo_flags &= ~MO_SET;
2366                 if (mop->mo_arg != NULL) {
2367                         kmem_free(mop->mo_arg, strlen(mop->mo_arg) + 1);
2368                 }
2369                 mop->mo_arg = NULL;
2370                 if (update_mnttab)
2371                         vfs_mnttab_modtimeupd();
```

```
2372                    break;
2373            }
2374    }

2376    void
2377    vfs_clearmntopt(struct vfs *vfsp, const char *opt)
2378    {
2379            int gotlock = 0;

2381            if (VFS_ON_LIST(vfsp)) {
2382                    gotlock = 1;
2383                    vfs_list_lock();
2384            }
2385            vfs_clearmntopt_nolock(&vfsp->vfs_mntopts, opt, gotlock);
2386            if (gotlock)
2387                    vfs_list_unlock();
2388    }


2391    /*
2392     * Set a mount option on.  If it's not found in the table, it's silently
2393     * ignored.  If the option has MO_IGNORE set, it is still set unless the
2394     * VFS_NOFORCEOPT bit is set in the flags.  Also, VFS_DISPLAY/VFS_NODISPLAY flag
2395     * bits can be used to toggle the MO_NODISPLAY bit for the option.
2396     * If the VFS_CREATEOPT flag bit is set then the first option slot with
2397     * MO_EMPTY set is created as the option passed in.
2398     *
2399     * The update_mnttab arg indicates whether mops is part of a vfs that is on
2400     * the vfs list.
2401     */
2402    static void
2403    vfs_setmntopt_nolock(mntopts_t *mops, const char *opt,
2404        const char *arg, int flags, int update_mnttab)
2405    {
2406            mntopt_t *mop;
2407            uint_t i, count;
2408            char *sp;

2410            ASSERT(!update_mnttab || RW_WRITE_HELD(&vfslist));

2412            if (flags & VFS_CREATEOPT) {
2413                    if (vfs_hasopt(mops, opt) != NULL) {
2414                            flags &= ~VFS_CREATEOPT;
2415                    }
2416            }
2417            count = mops->mo_count;
2418            for (i = 0; i < count; i++) {
2419                    mop = &mops->mo_list[i];

2421                    if (mop->mo_flags & MO_EMPTY) {
2422                            if ((flags & VFS_CREATEOPT) == 0)
2423                                    continue;
2424                            sp = kmem_alloc(strlen(opt) + 1, KM_SLEEP);
2425                            (void) strcpy(sp, opt);
2426                            mop->mo_name = sp;
2427                            if (arg != NULL)
2428                                    mop->mo_flags = MO_HASVALUE;
2429                            else
2430                                    mop->mo_flags = 0;
2431                    } else if (strcmp(opt, mop->mo_name)) {
2432                            continue;
2433                    }
2434                    if ((mop->mo_flags & MO_IGNORE) && (flags & VFS_NOFORCEOPT))
2435                            break;
2436                    if (arg != NULL && (mop->mo_flags & MO_HASVALUE) != 0) {
2437                            sp = kmem_alloc(strlen(arg) + 1, KM_SLEEP);
```



```
2438                            (void) strcpy(sp, arg);
2439                    } else {
2440                            sp = NULL;
2441                    }
2442                    if (mop->mo_arg != NULL)
2443                            kmem_free(mop->mo_arg, strlen(mop->mo_arg) + 1);
2444                    mop->mo_arg = sp;
2445                    if (flags & VFS_DISPLAY)
2446                            mop->mo_flags &= ~MO_NODISPLAY;
2447                    if (flags & VFS_NODISPLAY)
2448                            mop->mo_flags |= MO_NODISPLAY;
2449                    mop->mo_flags |= MO_SET;
2450                    if (mop->mo_cancel != NULL) {
2451                            char **cp;

2453                            for (cp = mop->mo_cancel; *cp != NULL; cp++)
2454                                    vfs_clearmntopt_nolock(mops, *cp, 0);
2455                    }
2456                    if (update_mnttab)
2457                            vfs_mnttab_modtimeupd();
2458                    break;
2459            }
2460    }

2462    void
2463    vfs_setmntopt(struct vfs *vfsp, const char *opt, const char *arg, int flags)
2464    {
2465            int gotlock = 0;

2467            if (VFS_ON_LIST(vfsp)) {
2468                    gotlock = 1;
2469                    vfs_list_lock();
2470            }
2471            vfs_setmntopt_nolock(&vfsp->vfs_mntopts, opt, arg, flags, gotlock);
2472            if (gotlock)
2473                    vfs_list_unlock();
2474    }


2477    /*
2478     * Add a "tag" option to a mounted file system's options list.
2479     *
2480     * Note: caller is responsible for locking the vfs list, if needed,
2481     *       to protect mops.
2482     */
2483    static mntopt_t *
2484    vfs_addtag(mntopts_t *mops, const char *tag)
2485    {
2486            uint_t count;
2487            mntopt_t *mop, *motbl;

2489            count = mops->mo_count + 1;
2490            motbl = kmem_zalloc(count * sizeof (mntopt_t), KM_SLEEP);
2491            if (mops->mo_count) {
2492                    size_t len = (count - 1) * sizeof (mntopt_t);

2494                    bcopy(mops->mo_list, motbl, len);
2495                    kmem_free(mops->mo_list, len);
2496            }
2497            mops->mo_count = count;
2498            mops->mo_list = motbl;
2499            mop = &motbl[count - 1];
2500            mop->mo_flags = MO_TAG;
2501            mop->mo_name = kmem_alloc(strlen(tag) + 1, KM_SLEEP);
2502            (void) strcpy(mop->mo_name, tag);
2503            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;
```

```
2636         int setflg = VFS_NOFORCEOPT;

2638         if (osp == NULL)
2639                 return;
2640         while (*s != '\0') {
2641                 p = strchr(s, ',');     /* find next option */
2642                 if (p == NULL) {
2643                         cp = NULL;
2644                         p = s + strlen(s);
2645                 } else {
2646                         cp = p;         /* save location of comma */
2647                         *p++ = '\0';    /* mark end and point to next option */
2648                 }
2649                 nextop = p;
2650                 p = strchr(s, '=');     /* look for value */
2651                 if (p == NULL) {
2652                         valp = NULL;    /* no value supplied */
2653                 } else {
2654                         ep = p;         /* save location of equals */
2655                         *p++ = '\0';    /* end option and point to value */
2656                         valp = p;
2657                 }
2658                 /*
2659                  * set option into options table
2660                  */
2661                 if (create)
2662                         setflg |= VFS_CREATEOPT;
2663                 vfs_setmntopt_nolock(mops, s, valp, setflg, 0);
2664                 if (cp != NULL)
2665                         *cp = ',';      /* restore the comma */
2666                 if (valp != NULL)
2667                         *ep = '=';      /* restore the equals */
2668                 s = nextop;
2669         }
2670 }

2672 /*
2673  * Function to inquire if an option exists in a mount options table.
2674  * Returns a pointer to the option if it exists, else NULL.
2675  *
2676  * This function is *not* for general use by filesystems.
2677  *
2678  * Note: caller is responsible for locking the vfs list, if needed,
2679  *       to protect mops.
2680  */
2681 struct mntopt *
2682 vfs_hasopt(const mntopts_t *mops, const char *opt)
2683 {
2684         struct mntopt *mop;
2685         uint_t i, count;

2687         count = mops->mo_count;
2688         for (i = 0; i < count; i++) {
2689                 mop = &mops->mo_list[i];

2691                 if (mop->mo_flags & MO_EMPTY)
2692                         continue;
2693                 if (strcmp(opt, mop->mo_name) == 0)
2694                         return (mop);
2695         }
2696         return (NULL);
2697 }

2699 /*
2700  * Function to inquire if an option is set in a mount options table.
2701  * Returns non-zero if set and fills in the arg pointer with a pointer to
```



```
2702  * the argument string or NULL if there is no argument string.
2703  */
2704 static int
2705 vfs_optionisset_nolock(const mntopts_t *mops, const char *opt, char **argp)
2706 {
2707         struct mntopt *mop;
2708         uint_t i, count;

2710         count = mops->mo_count;
2711         for (i = 0; i < count; i++) {
2712                 mop = &mops->mo_list[i];

2714                 if (mop->mo_flags & MO_EMPTY)
2715                         continue;
2716                 if (strcmp(opt, mop->mo_name))
2717                         continue;
2718                 if ((mop->mo_flags & MO_SET) == 0)
2719                         return (0);
2720                 if (argp != NULL && (mop->mo_flags & MO_HASVALUE) != 0)
2721                         *argp = mop->mo_arg;
2722                 return (1);
2723         }
2724         return (0);
2725 }


2728 int
2729 vfs_optionisset(const struct vfs *vfsp, const char *opt, char **argp)
2730 {
2731         int ret;

2733         vfs_list_read_lock();
2734         ret = vfs_optionisset_nolock(&vfsp->vfs_mntopts, opt, argp);
2735         vfs_list_unlock();
2736         return (ret);
2737 }


2740 /*
2741  * Construct a comma separated string of the options set in the given
2742  * mount table, return the string in the given buffer.  Return non-zero if
2743  * the buffer would overflow.
2744  *
2745  * This function is *not* for general use by filesystems.
2746  *
2747  * Note: caller is responsible for locking the vfs list, if needed,
2748  *       to protect mp.
2749  */
2750 int
2751 vfs_buildoptionstr(const mntopts_t *mp, char *buf, int len)
2752 {
2753         char *cp;
2754         uint_t i;

2756         buf[0] = '\0';
2757         cp = buf;
2758         for (i = 0; i < mp->mo_count; i++) {
2759                 struct mntopt *mop;

2761                 mop = &mp->mo_list[i];
2762                 if (mop->mo_flags & MO_SET) {
2763                         int optlen, comma = 0;

2765                         if (buf[0] != '\0')
2766                                 comma = 1;
2767                         optlen = strlen(mop->mo_name);
```

```
2768                        if (strlen(buf) + comma + optlen + 1 > len)
2769                                goto err;
2770                        if (comma)
2771                                *cp++ = ',';
2772                        (void) strcpy(cp, mop->mo_name);
2773                        cp += optlen;
2774                        /*
2775                         * Append option value if there is one
2776                         */
2777                        if (mop->mo_arg != NULL) {
2778                                int arglen;

2780                                arglen = strlen(mop->mo_arg);
2781                                if (strlen(buf) + arglen + 2 > len)
2782                                        goto err;
2783                                *cp++ = '=';
2784                                (void) strcpy(cp, mop->mo_arg);
2785                                cp += arglen;
2786                        }
2787                }
2788        }
2789        return (0);
2790 err:
2791        return (EOVERFLOW);
2792 }

2794 static void
2795 vfs_freecancelopt(char **moc)
2796 {
2797        if (moc != NULL) {
2798                int ccnt = 0;
2799                char **cp;

2801                for (cp = moc; *cp != NULL; cp++) {
2802                        kmem_free(*cp, strlen(*cp) + 1);
2803                        ccnt++;
2804                }
2805                kmem_free(moc, (ccnt + 1) * sizeof (char *));
2806        }
2807 }

2809 static void
2810 vfs_freeopt(mntopt_t *mop)
2811 {
2812        if (mop->mo_name != NULL)
2813                kmem_free(mop->mo_name, strlen(mop->mo_name) + 1);

2815        vfs_freecancelopt(mop->mo_cancel);

2817        if (mop->mo_arg != NULL)
2818                kmem_free(mop->mo_arg, strlen(mop->mo_arg) + 1);
2819 }

2821 /*
2822  * Free a mount options table
2823  *
2824  * This function is *not* for general use by filesystems.
2825  *
2826  * Note: caller is responsible for locking the vfs list, if needed,
2827  *       to protect mp.
2828  */
2829 void
2830 vfs_freeopttbl(mntopts_t *mp)
2831 {
2832        uint_t i, count;
```



```
2834        count = mp->mo_count;
2835        for (i = 0; i < count; i++) {
2836                vfs_freeopt(&mp->mo_list[i]);
2837        }
2838        if (count) {
2839                kmem_free(mp->mo_list, sizeof (mntopt_t) * count);
2840                mp->mo_count = 0;
2841                mp->mo_list = NULL;
2842        }
2843 }


2846 /* ARGSUSED */
2847 static int
2848 vfs_mntdummyread(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cred,
2849        caller_context_t *ct)
2850 {
2851        return (0);
2852 }

2854 /* ARGSUSED */
2855 static int
2856 vfs_mntdummywrite(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cred,
2857        caller_context_t *ct)
2858 {
2859        return (0);
2860 }

2862 /*
2863  * The dummy vnode is currently used only by file events notification
2864  * module which is just interested in the timestamps.
2865  */
2866 /* ARGSUSED */
2867 static int
2868 vfs_mntdummygetattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2869     caller_context_t *ct)
2870 {
2871        bzero(vap, sizeof (vattr_t));
2872        vap->va_type = VREG;
2873        vap->va_nlink = 1;
2874        vap->va_ctime = vfs_mnttab_ctime;
2875        /*
2876         * it is ok to just copy mtime as the time will be monotonically
2877         * increasing.
2878         */
2879        vap->va_mtime = vfs_mnttab_mtime;
2880        vap->va_atime = vap->va_mtime;
2881        return (0);
2882 }

2884 static void
2885 vfs_mnttabvp_setup(void)
2886 {
2887        vnode_t *tvp;
2888        vnodeops_t *vfs_mntdummyvnops;
2889        const fs_operation_def_t mnt_dummyvnodeops_template[] = {
2890                VOPNAME_READ,           { .vop_read = vfs_mntdummyread },
2891                VOPNAME_WRITE,          { .vop_write = vfs_mntdummywrite },
2892                VOPNAME_GETATTR,        { .vop_getattr = vfs_mntdummygetattr },
2893                VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
2894                NULL,                   NULL
2895        };

2897        if (vn_make_ops("mnttab", mnt_dummyvnodeops_template,
2898            &vfs_mntdummyvnops) != 0) {
2899                cmn_err(CE_WARN, "vfs_mnttabvp_setup: vn_make_ops failed");
```

```
2900                 /* Shouldn't happen, but not bad enough to panic */
2901                 return;
2902         }

2904         /*
2905          * A global dummy vnode is allocated to represent mntfs files.
2906          * The mntfs file (/etc/mnttab) can be monitored for file events
2907          * and receive an event when mnttab changes. Dummy VOP calls
2908          * will be made on this vnode. The file events notification module
2909          * intercepts this vnode and delivers relevant events.
2910          */
2911         tvp = vn_alloc(KM_SLEEP);
2912         tvp->v_flag = VNOMOUNT|VNOMAP|VNOSWAP|VNOCACHE;
2913         vn_setops(tvp, vfs_mntdummyvnops);
2914         tvp->v_type = VREG;
2915         /*
2916          * The mnt dummy ops do not reference v_data.
2917          * No other module intercepting this vnode should either.
2918          * Just set it to point to itself.
2919          */
2920         tvp->v_data = (caddr_t)tvp;
2921         tvp->v_vfsp = rootvfs;
2922         vfs_mntdummyvp = tvp;
2923 }

2925 /*
2926  * performs fake read/write ops
2927  */
2928 static void
2929 vfs_mnttab_rwop(int rw)
2930 {
2931         struct uio      uio;
2932         struct iovec    iov;
2933         char    buf[1];

2935         if (vfs_mntdummyvp == NULL)
2936                 return;

2938         bzero(&uio, sizeof (uio));
2939         bzero(&iov, sizeof (iov));
2940         iov.iov_base = buf;
2941         iov.iov_len = 0;
2942         uio.uio_iov = &iov;
2943         uio.uio_iovcnt = 1;
2944         uio.uio_loffset = 0;
2945         uio.uio_segflg = UIO_SYSSPACE;
2946         uio.uio_resid = 0;
2947         if (rw) {
2948                 (void) VOP_WRITE(vfs_mntdummyvp, &uio, 0, kcred, NULL);
2949         } else {
2950                 (void) VOP_READ(vfs_mntdummyvp, &uio, 0, kcred, NULL);
2951         }
2952 }

2954 /*
2955  * Generate a write operation.
2956  */
2957 void
2958 vfs_mnttab_writeop(void)
2959 {
2960         vfs_mnttab_rwop(1);
2961 }

2963 /*
2964  * Generate a read operation.
2965  */
```



```
2966 void
2967 vfs_mnttab_readop(void)
2968 {
2969         vfs_mnttab_rwop(0);
2970 }

2972 /*
2973  * Free any mnttab information recorded in the vfs struct.
2974  * The vfs must not be on the vfs list.
2975  */
2976 static void
2977 vfs_freemnttab(struct vfs *vfsp)
2978 {
2979         ASSERT(!VFS_ON_LIST(vfsp));

2981         /*
2982          * Free device and mount point information
2983          */
2984         if (vfsp->vfs_mntpt != NULL) {
2985                 refstr_rele(vfsp->vfs_mntpt);
2986                 vfsp->vfs_mntpt = NULL;
2987         }
2988         if (vfsp->vfs_resource != NULL) {
2989                 refstr_rele(vfsp->vfs_resource);
2990                 vfsp->vfs_resource = NULL;
2991         }
2992         /*
2993          * Now free mount options information
2994          */
2995         vfs_freeopttbl(&vfsp->vfs_mntopts);
2996 }

2998 /*
2999  * Return the last mnttab modification time
3000  */
3001 void
3002 vfs_mnttab_modtime(timespec_t *ts)
3003 {
3004         ASSERT(RW_LOCK_HELD(&vfslist));
3005         *ts = vfs_mnttab_mtime;
3006 }

3008 /*
3009  * See if mnttab is changed
3010  */
3011 void
3012 vfs_mnttab_poll(timespec_t *old, struct pollhead **phpp)
3013 {
3014         int changed;

3016         *phpp = (struct pollhead *)NULL;

3018         /*
3019          * Note: don't grab vfs list lock before accessing vfs_mnttab_mtime.
3020          * Can lead to deadlock against vfs_mnttab_modtimeupd(). It is safe
3021          * to not grab the vfs list lock because tv_sec is monotonically
3022          * increasing.
3023          */

3025         changed = (old->tv_nsec != vfs_mnttab_mtime.tv_nsec) ||
3026             (old->tv_sec != vfs_mnttab_mtime.tv_sec);
3027         if (!changed) {
3028                 *phpp = &vfs_pollhd;
3029         }
3030 }
```

```
3032 /* Provide a unique and monotonically-increasing timestamp. */
3033 void
3034 vfs_mono_time(timespec_t *ts)
3035 {
3036         static volatile hrtime_t hrt;           /* The saved time. */
3037         hrtime_t        newhrt, oldhrt;         /* For effecting the CAS. */
3038         timespec_t      newts;
3039
3040         /*
3041          * Try gethrestime() first, but be prepared to fabricate a sensible
3042          * answer at the first sign of any trouble.
3043          */
3044         gethrestime(&newts);
3045         newhrt = ts2hrt(&newts);
3046         for (;;) {
3047                 oldhrt = hrt;
3048                 if (newhrt <= hrt)
3049                         newhrt = hrt + 1;
3050                 if (atomic_cas_64((uint64_t *)&hrt, oldhrt, newhrt) == oldhrt)
3051                         break;
3052         }
3053         hrt2ts(newhrt, ts);
3054 }
3055
3056 /*
3057  * Update the mnttab modification time and wake up any waiters for
3058  * mnttab changes
3059  */
3060 void
3061 vfs_mnttab_modtimeupd()
3062 {
3063         hrtime_t oldhrt, newhrt;
3064
3065         ASSERT(RW_WRITE_HELD(&vfslist));
3066         oldhrt = ts2hrt(&vfs_mnttab_mtime);
3067         gethrestime(&vfs_mnttab_mtime);
3068         newhrt = ts2hrt(&vfs_mnttab_mtime);
3069         if (oldhrt == (hrtime_t)0)
3070                 vfs_mnttab_ctime = vfs_mnttab_mtime;
3071         /*
3072          * Attempt to provide unique mtime (like uniqtime but not).
3073          */
3074         if (newhrt == oldhrt) {
3075                 newhrt++;
3076                 hrt2ts(newhrt, &vfs_mnttab_mtime);
3077         }
3078         pollwakeup(&vfs_pollhd, (short)POLLRDBAND);
3079         vfs_mnttab_writeop();
3080 }
3081
3082 int
3083 dounmount(struct vfs *vfsp, int flag, cred_t *cr)
3084 {
3085         vnode_t *coveredvp;
3086         int error;
3087         extern void teardown_vopstats(vfs_t *);
3088
3089         /*
3090          * Get covered vnode. This will be NULL if the vfs is not linked
3091          * into the file system name space (i.e., domount() with MNT_NOSPICE).
3092          */
3093         coveredvp = vfsp->vfs_vnodecovered;
3094         ASSERT(coveredvp == NULL || vn_vfswlock_held(coveredvp));
3095
3096         /*
3097          * Purge all dnlc entries for this vfs.
```



```
3098          */
3099         (void) dnlc_purge_vfsp(vfsp, 0);
3100
3101         /* For forcible umount, skip VFS_SYNC() since it may hang */
3102         if ((flag & MS_FORCE) == 0)
3103                 (void) VFS_SYNC(vfsp, 0, cr);
3104
3105         /*
3106          * Lock the vfs to maintain fs status quo during unmount.  This
3107          * has to be done after the sync because ufs_update tries to acquire
3108          * the vfs_reflock.
3109          */
3110         vfs_lock_wait(vfsp);
3111
3112         if (error = VFS_UNMOUNT(vfsp, flag, cr)) {
3113                 vfs_unlock(vfsp);
3114                 if (coveredvp != NULL)
3115                         vn_vfsunlock(coveredvp);
3116         } else if (coveredvp != NULL) {
3117                 teardown_vopstats(vfsp);
3118                 /*
3119                  * vfs_remove() will do a VN_RELE(vfsp->vfs_vnodecovered)
3120                  * when it frees vfsp so we do a VN_HOLD() so we can
3121                  * continue to use coveredvp afterwards.
3122                  */
3123                 VN_HOLD(coveredvp);
3124                 vfs_remove(vfsp);
3125                 vn_vfsunlock(coveredvp);
3126                 VN_RELE(coveredvp);
3127         } else {
3128                 teardown_vopstats(vfsp);
3129                 /*
3130                  * Release the reference to vfs that is not linked
3131                  * into the name space.
3132                  */
3133                 vfs_unlock(vfsp);
3134                 VFS_RELE(vfsp);
3135         }
3136         return (error);
3137 }
3138
3139
3140 /*
3141  * Vfs_unmountall() is called by uadmin() to unmount all
3142  * mounted file systems (except the root file system) during shutdown.
3143  * It follows the existing locking protocol when traversing the vfs list
3144  * to sync and unmount vfses. Even though there should be no
3145  * other thread running while the system is shutting down, it is prudent
3146  * to still follow the locking protocol.
3147  */
3148 void
3149 vfs_unmountall(void)
3150 {
3151         struct vfs *vfsp;
3152         struct vfs *prev_vfsp = NULL;
3153         int error;
3154
3155         /*
3156          * Toss all dnlc entries now so that the per-vfs sync
3157          * and unmount operations don't have to slog through
3158          * a bunch of uninteresting vnodes over and over again.
3159          */
3160         dnlc_purge();
3161
3162         vfs_list_lock();
3163         for (vfsp = rootvfs->vfs_prev; vfsp != rootvfs; vfsp = prev_vfsp) {
```

```
3164                 prev_vfsp = vfsp->vfs_prev;

3166                 if (vfs_lock(vfsp) != 0)
3167                         continue;
3168                 error = vn_vfswlock(vfsp->vfs_vnodecovered);
3169                 vfs_unlock(vfsp);
3170                 if (error)
3171                         continue;

3173                 vfs_list_unlock();

3175                 (void) VFS_SYNC(vfsp, SYNC_CLOSE, CRED());
3176                 (void) dounmount(vfsp, 0, CRED());

3178                 /*
3179                  * Since we dropped the vfslist lock above we must
3180                  * verify that next_vfsp still exists, else start over.
3181                  */
3182                 vfs_list_lock();
3183                 for (vfsp = rootvfs->vfs_prev;
3184                     vfsp != rootvfs; vfsp = vfsp->vfs_prev)
3185                         if (vfsp == prev_vfsp)
3186                                 break;
3187                 if (vfsp == rootvfs && prev_vfsp != rootvfs)
3188                         prev_vfsp = rootvfs->vfs_prev;
3189         }
3190         vfs_list_unlock();
3191 }

3193 /*
3194  * Called to add an entry to the end of the vfs mount in progress list
3195  */
3196 void
3197 vfs_addmip(dev_t dev, struct vfs *vfsp)
3198 {
3199         struct ipmnt *mipp;

3201         mipp = (struct ipmnt *)kmem_alloc(sizeof (struct ipmnt), KM_SLEEP);
3202         mipp->mip_next = NULL;
3203         mipp->mip_dev = dev;
3204         mipp->mip_vfsp = vfsp;
3205         mutex_enter(&vfs_miplist_mutex);
3206         if (vfs_miplist_end != NULL)
3207                 vfs_miplist_end->mip_next = mipp;
3208         else
3209                 vfs_miplist = mipp;
3210         vfs_miplist_end = mipp;
3211         mutex_exit(&vfs_miplist_mutex);
3212 }

3214 /*
3215  * Called to remove an entry from the mount in progress list
3216  * Either because the mount completed or it failed.
3217  */
3218 void
3219 vfs_delmip(struct vfs *vfsp)
3220 {
3221         struct ipmnt *mipp, *mipprev;

3223         mutex_enter(&vfs_miplist_mutex);
3224         mipprev = NULL;
3225         for (mipp = vfs_miplist;
3226             mipp && mipp->mip_vfsp != vfsp; mipp = mipp->mip_next) {
3227                 mipprev = mipp;
3228         }
3229         if (mipp == NULL)
```



```
3230                 return; /* shouldn't happen */
3231         if (mipp == vfs_miplist_end)
3232                 vfs_miplist_end = mipprev;
3233         if (mipprev == NULL)
3234                 vfs_miplist = mipp->mip_next;
3235         else
3236                 mipprev->mip_next = mipp->mip_next;
3237         mutex_exit(&vfs_miplist_mutex);
3238         kmem_free(mipp, sizeof (struct ipmnt));
3239 }

3241 /*
3242  * vfs_add is called by a specific filesystem's mount routine to add
3243  * the new vfs into the vfs list/hash and to cover the mounted-on vnode.
3244  * The vfs should already have been locked by the caller.
3245  *
3246  * coveredvp is NULL if this is the root.
3247  */
3248 void
3249 vfs_add(vnode_t *coveredvp, struct vfs *vfsp, int mflag)
3250 {
3251         int newflag;

3253         ASSERT(vfs_lock_held(vfsp));
3254         VFS_HOLD(vfsp);
3255         newflag = vfsp->vfs_flag;
3256         if (mflag & MS_RDONLY)
3257                 newflag |= VFS_RDONLY;
3258         else
3259                 newflag &= ~VFS_RDONLY;
3260         if (mflag & MS_NOSUID)
3261                 newflag |= (VFS_NOSETUID|VFS_NODEVICES);
3262         else
3263                 newflag &= ~(VFS_NOSETUID|VFS_NODEVICES);
3264         if (mflag & MS_NOMNTTAB)
3265                 newflag |= VFS_NOMNTTAB;
3266         else
3267                 newflag &= ~VFS_NOMNTTAB;

3269         if (coveredvp != NULL) {
3270                 ASSERT(vn_vfswlock_held(coveredvp));
3271                 coveredvp->v_vfsmountedhere = vfsp;
3272                 VN_HOLD(coveredvp);
3273         }
3274         vfsp->vfs_vnodecovered = coveredvp;
3275         vfsp->vfs_flag = newflag;

3277         vfs_list_add(vfsp);
3278 }

3280 /*
3281  * Remove a vfs from the vfs list, null out the pointer from the
3282  * covered vnode to the vfs (v_vfsmountedhere), and null out the pointer
3283  * from the vfs to the covered vnode (vfs_vnodecovered). Release the
3284  * reference to the vfs and to the covered vnode.
3285  *
3286  * Called from dounmount after it's confirmed with the file system
3287  * that the unmount is legal.
3288  */
3289 void
3290 vfs_remove(struct vfs *vfsp)
3291 {
3292         vnode_t *vp;

3294         ASSERT(vfs_lock_held(vfsp));
```

```
3296         /*
3297          * Can't unmount root.  Should never happen because fs will
3298          * be busy.
3299          */
3300         if (vfsp == rootvfs)
3301                 panic("vfs_remove: unmounting root");

3303         vfs_list_remove(vfsp);

3305         /*
3306          * Unhook from the file system name space.
3307          */
3308         vp = vfsp->vfs_vnodecovered;
3309         ASSERT(vn_vfswlock_held(vp));
3310         vp->v_vfsmountedhere = NULL;
3311         vfsp->vfs_vnodecovered = NULL;
3312         VN_RELE(vp);

3314         /*
3315          * Release lock and wakeup anybody waiting.
3316          */
3317         vfs_unlock(vfsp);
3318         VFS_RELE(vfsp);
3319 }

3321 /*
3322  * Lock a filesystem to prevent access to it while mounting,
3323  * unmounting and syncing.  Return EBUSY immediately if lock
3324  * can't be acquired.
3325  */
3326 int
3327 vfs_lock(vfs_t *vfsp)
3328 {
3329         vn_vfslocks_entry_t *vpvfsentry;

3331         vpvfsentry = vn_vfslocks_getlock(vfsp);
3332         if (rwst_tryenter(&vpvfsentry->ve_lock, RW_WRITER))
3333                 return (0);

3335         vn_vfslocks_rele(vpvfsentry);
3336         return (EBUSY);
3337 }

3339 int
3340 vfs_rlock(vfs_t *vfsp)
3341 {
3342         vn_vfslocks_entry_t *vpvfsentry;

3344         vpvfsentry = vn_vfslocks_getlock(vfsp);

3346         if (rwst_tryenter(&vpvfsentry->ve_lock, RW_READER))
3347                 return (0);

3349         vn_vfslocks_rele(vpvfsentry);
3350         return (EBUSY);
3351 }

3353 void
3354 vfs_lock_wait(vfs_t *vfsp)
3355 {
3356         vn_vfslocks_entry_t *vpvfsentry;

3358         vpvfsentry = vn_vfslocks_getlock(vfsp);
3359         rwst_enter(&vpvfsentry->ve_lock, RW_WRITER);
3360 }
```



```
3362 void
3363 vfs_rlock_wait(vfs_t *vfsp)
3364 {
3365         vn_vfslocks_entry_t *vpvfsentry;

3367         vpvfsentry = vn_vfslocks_getlock(vfsp);
3368         rwst_enter(&vpvfsentry->ve_lock, RW_READER);
3369 }

3371 /*
3372  * Unlock a locked filesystem.
3373  */
3374 void
3375 vfs_unlock(vfs_t *vfsp)
3376 {
3377         vn_vfslocks_entry_t *vpvfsentry;

3379         /*
3380          * vfs_unlock will mimic sema_v behaviour to fix 4748018.
3381          * And these changes should remain for the patch changes as it is.
3382          */
3383         if (panicstr)
3384                 return;

3386         /*
3387          * ve_refcount needs to be dropped twice here.
3388          * 1. To release refernce after a call to vfs_locks_getlock()
3389          * 2. To release the reference from the locking routines like
3390          *    vfs_rlock_wait/vfs_wlock_wait/vfs_wlock etc,.
3391          */

3393         vpvfsentry = vn_vfslocks_getlock(vfsp);
3394         vn_vfslocks_rele(vpvfsentry);

3396         rwst_exit(&vpvfsentry->ve_lock);
3397         vn_vfslocks_rele(vpvfsentry);
3398 }

3400 /*
3401  * Utility routine that allows a filesystem to construct its
3402  * fsid in "the usual way" - by munging some underlying dev_t and
3403  * the filesystem type number into the 64-bit fsid.  Note that
3404  * this implicitly relies on dev_t persistence to make filesystem
3405  * id's persistent.
3406  *
3407  * There's nothing to prevent an individual fs from constructing its
3408  * fsid in a different way, and indeed they should.
3409  *
3410  * Since we want fsids to be 32-bit quantities (so that they can be
3411  * exported identically by either 32-bit or 64-bit APIs, as well as
3412  * the fact that fsid's are "known" to NFS), we compress the device
3413  * number given down to 32-bits, and panic if that isn't possible.
3414  */
3415 void
3416 vfs_make_fsid(fsid_t *fsi, dev_t dev, int val)
3417 {
3418         if (!cmpldev((dev32_t *)&fsi->val[0], dev))
3419                 panic("device number too big for fsid!");
3420         fsi->val[1] = val;
3421 }

3423 int
3424 vfs_lock_held(vfs_t *vfsp)
3425 {
3426         int held;
3427         vn_vfslocks_entry_t *vpvfsentry;
```

```
3429         /*
3430          * vfs_lock_held will mimic sema_held behaviour
3431          * if panicstr is set. And these changes should remain
3432          * for the patch changes as it is.
3433          */
3434         if (panicstr)
3435                 return (1);

3437         vpvfsentry = vn_vfslocks_getlock(vfsp);
3438         held = rwst_lock_held(&vpvfsentry->ve_lock, RW_WRITER);

3440         vn_vfslocks_rele(vpvfsentry);
3441         return (held);
3442 }

3444 struct _kthread *
3445 vfs_lock_owner(vfs_t *vfsp)
3446 {
3447         struct _kthread *owner;
3448         vn_vfslocks_entry_t *vpvfsentry;

3450         /*
3451          * vfs_wlock_held will mimic sema_held behaviour
3452          * if panicstr is set. And these changes should remain
3453          * for the patch changes as it is.
3454          */
3455         if (panicstr)
3456                 return (NULL);

3458         vpvfsentry = vn_vfslocks_getlock(vfsp);
3459         owner = rwst_owner(&vpvfsentry->ve_lock);

3461         vn_vfslocks_rele(vpvfsentry);
3462         return (owner);
3463 }

3465 /*
3466  * vfs list locking.
3467  *
3468  * Rather than manipulate the vfslist lock directly, we abstract into lock
3469  * and unlock routines to allow the locking implementation to be changed for
3470  * clustering.
3471  *
3472  * Whenever the vfs list is modified through its hash links, the overall list
3473  * lock must be obtained before locking the relevant hash bucket.  But to see
3474  * whether a given vfs is on the list, it suffices to obtain the lock for the
3475  * hash bucket without getting the overall list lock.  (See getvfs() below.)
3476  */

3478 void
3479 vfs_list_lock()
3480 {
3481         rw_enter(&vfslist, RW_WRITER);
3482 }

3484 void
3485 vfs_list_read_lock()
3486 {
3487         rw_enter(&vfslist, RW_READER);
3488 }

3490 void
3491 vfs_list_unlock()
3492 {
3493         rw_exit(&vfslist);
```



```
3494 }

3496 /*
3497  * Low level worker routines for adding entries to and removing entries from
3498  * the vfs list.
3499  */

3501 static void
3502 vfs_hash_add(struct vfs *vfsp, int insert_at_head)
3503 {
3504         int vhno;
3505         struct vfs **hp;
3506         dev_t dev;

3508         ASSERT(RW_WRITE_HELD(&vfslist));

3510         dev = expldev(vfsp->vfs_fsid.val[0]);
3511         vhno = VFSHASH(getmajor(dev), getminor(dev));

3513         mutex_enter(&rvfs_list[vhno].rvfs_lock);

3515         /*
3516          * Link into the hash table, inserting it at the end, so that LOFS
3517          * with the same fsid as UFS (or other) file systems will not hide the
3518          * UFS.
3519          */
3520         if (insert_at_head) {
3521                 vfsp->vfs_hash = rvfs_list[vhno].rvfs_head;
3522                 rvfs_list[vhno].rvfs_head = vfsp;
3523         } else {
3524                 for (hp = &rvfs_list[vhno].rvfs_head; *hp != NULL;
3525                     hp = &(*hp)->vfs_hash)
3526                         continue;
3527                 /*
3528                  * hp now contains the address of the pointer to update
3529                  * to effect the insertion.
3530                  */
3531                 vfsp->vfs_hash = NULL;
3532                 *hp = vfsp;
3533         }

3535         rvfs_list[vhno].rvfs_len++;
3536         mutex_exit(&rvfs_list[vhno].rvfs_lock);
3537 }


3540 static void
3541 vfs_hash_remove(struct vfs *vfsp)
3542 {
3543         int vhno;
3544         struct vfs *tvfsp;
3545         dev_t dev;

3547         ASSERT(RW_WRITE_HELD(&vfslist));

3549         dev = expldev(vfsp->vfs_fsid.val[0]);
3550         vhno = VFSHASH(getmajor(dev), getminor(dev));

3552         mutex_enter(&rvfs_list[vhno].rvfs_lock);

3554         /*
3555          * Remove from hash.
3556          */
3557         if (rvfs_list[vhno].rvfs_head == vfsp) {
3558                 rvfs_list[vhno].rvfs_head = vfsp->vfs_hash;
3559                 rvfs_list[vhno].rvfs_len--;
```

```
3560                goto foundit;
3561        }
3562        for (tvfsp = rvfs_list[vhno].rvfs_head; tvfsp != NULL;
3563            tvfsp = tvfsp->vfs_hash) {
3564                if (tvfsp->vfs_hash == vfsp) {
3565                        tvfsp->vfs_hash = vfsp->vfs_hash;
3566                        rvfs_list[vhno].rvfs_len--;
3567                        goto foundit;
3568                }
3569        }
3570        cmn_err(CE_WARN, "vfs_list_remove: vfs not found in hash");

3572 foundit:

3574        mutex_exit(&rvfs_list[vhno].rvfs_lock);
3575 }


3578 void
3579 vfs_list_add(struct vfs *vfsp)
3580 {
3581        zone_t *zone;

3583        /*
3584         * Typically, the vfs_t will have been created on behalf of the file
3585         * system in vfs_init, where it will have been provided with a
3586         * vfs_impl_t. This, however, might be lacking if the vfs_t was created
3587         * by an unbundled file system. We therefore check for such an example
3588         * before stamping the vfs_t with its creation time for the benefit of
3589         * mntfs.
3590         */
3591        if (vfsp->vfs_implp == NULL)
3592                vfsimpl_setup(vfsp);
3593        vfs_mono_time(&vfsp->vfs_hrctime);

3595        /*
3596         * The zone that owns the mount is the one that performed the mount.
3597         * Note that this isn't necessarily the same as the zone mounted into.
3598         * The corresponding zone_rele_ref() will be done when the vfs_t
3599         * is being free'd.
3600         */
3601        vfsp->vfs_zone = curproc->p_zone;
3602        zone_init_ref(&vfsp->vfs_implp->vi_zone_ref);
3603        zone_hold_ref(vfsp->vfs_zone, &vfsp->vfs_implp->vi_zone_ref,
3604            ZONE_REF_VFS);

3606        /*
3607         * Find the zone mounted into, and put this mount on its vfs list.
3608         */
3609        zone = zone_find_by_path(refstr_value(vfsp->vfs_mntpt));
3610        ASSERT(zone != NULL);
3611        /*
3612         * Special casing for the root vfs.  This structure is allocated
3613         * statically and hooked onto rootvfs at link time.  During the
3614         * vfs_mountroot call at system startup time, the root file system's
3615         * VFS_MOUNTROOT routine will call vfs_add with this root vfs struct
3616         * as argument.  The code below must detect and handle this special
3617         * case.  The only apparent justification for this special casing is
3618         * to ensure that the root file system appears at the head of the
3619         * list.
3620         *
3621         * XXX: I'm assuming that it's ok to do normal list locking when
3622         *      adding the entry for the root file system (this used to be
3623         *      done with no locks held).
3624         */
3625        vfs_list_lock();
```



```
3626        /*
3627         * Link into the vfs list proper.
3628         */
3629        if (vfsp == &root) {
3630                /*
3631                 * Assert: This vfs is already on the list as its first entry.
3632                 * Thus, there's nothing to do.
3633                 */
3634                ASSERT(rootvfs == vfsp);
3635                /*
3636                 * Add it to the head of the global zone's vfslist.
3637                 */
3638                ASSERT(zone == global_zone);
3639                ASSERT(zone->zone_vfslist == NULL);
3640                zone->zone_vfslist = vfsp;
3641        } else {
3642                /*
3643                 * Link to end of list using vfs_prev (as rootvfs is now a
3644                 * doubly linked circular list) so list is in mount order for
3645                 * mnttab use.
3646                 */
3647                rootvfs->vfs_prev->vfs_next = vfsp;
3648                vfsp->vfs_prev = rootvfs->vfs_prev;
3649                rootvfs->vfs_prev = vfsp;
3650                vfsp->vfs_next = rootvfs;

3652                /*
3653                 * Do it again for the zone-private list (which may be NULL).
3654                 */
3655                if (zone->zone_vfslist == NULL) {
3656                        ASSERT(zone != global_zone);
3657                        zone->zone_vfslist = vfsp;
3658                } else {
3659                        zone->zone_vfslist->vfs_zone_prev->vfs_zone_next = vfsp;
3660                        vfsp->vfs_zone_prev = zone->zone_vfslist->vfs_zone_prev;
3661                        zone->zone_vfslist->vfs_zone_prev = vfsp;
3662                        vfsp->vfs_zone_next = zone->zone_vfslist;
3663                }
3664        }

3666        /*
3667         * Link into the hash table, inserting it at the end, so that LOFS
3668         * with the same fsid as UFS (or other) file systems will not hide
3669         * the UFS.
3670         */
3671        vfs_hash_add(vfsp, 0);

3673        /*
3674         * Link into tree indexed by mntpoint, for vfs_mntpoint2vfsp
3675         * mntix discerns entries with the same key
3676         */
3677        vfsp->vfs_mntix = ++vfs_curr_mntix;
3678        avl_add(&vfs_by_dev, vfsp);

3680        /*
3681         * Link into tree indexed by dev, for vfs_devismounted
3682         */
3683        avl_add(&vfs_by_mntpnt, vfsp);

3685        /*
3686 #endif /* ! codereview */
3687         * update the mnttab modification time
3688         */
3689        vfs_mnttab_modtimeupd();
3690        vfs_list_unlock();
3691        zone_rele(zone);
```

```
3692 }

3694 void
3695 vfs_list_remove(struct vfs *vfsp)
3696 {
3697         zone_t *zone;

3699         zone = zone_find_by_path(refstr_value(vfsp->vfs_mntpt));
3700         ASSERT(zone != NULL);
3701         /*
3702          * Callers are responsible for preventing attempts to unmount the
3703          * root.
3704          */
3705         ASSERT(vfsp != rootvfs);

3707         vfs_list_lock();

3709         /*
3710          * Remove from avl trees
3711          */
3712         avl_remove(&vfs_by_mntpnt, vfsp);
3713         avl_remove(&vfs_by_dev, vfsp);

3715         /*
3716 #endif /* ! codereview */
3717          * Remove from hash.
3718          */
3719         vfs_hash_remove(vfsp);

3721         /*
3722          * Remove from vfs list.
3723          */
3724         vfsp->vfs_prev->vfs_next = vfsp->vfs_next;
3725         vfsp->vfs_next->vfs_prev = vfsp->vfs_prev;
3726         vfsp->vfs_next = vfsp->vfs_prev = NULL;

3728         /*
3729          * Remove from zone-specific vfs list.
3730          */
3731         if (zone->zone_vfslist == vfsp)
3732                 zone->zone_vfslist = vfsp->vfs_zone_next;

3734         if (vfsp->vfs_zone_next == vfsp) {
3735                 ASSERT(vfsp->vfs_zone_prev == vfsp);
3736                 ASSERT(zone->zone_vfslist == vfsp);
3737                 zone->zone_vfslist = NULL;
3738         }

3740         vfsp->vfs_zone_prev->vfs_zone_next = vfsp->vfs_zone_next;
3741         vfsp->vfs_zone_next->vfs_zone_prev = vfsp->vfs_zone_prev;
3742         vfsp->vfs_zone_next = vfsp->vfs_zone_prev = NULL;

3744         /*
3745          * update the mnttab modification time
3746          */
3747         vfs_mnttab_modtimeupd();
3748         vfs_list_unlock();
3749         zone_rele(zone);
3750 }

3752 struct vfs *
3753 getvfs(fsid_t *fsid)
3754 {
3755         struct vfs *vfsp;
3756         int val0 = fsid->val[0];
3757         int val1 = fsid->val[1];
```



```
3758         dev_t dev = expldev(val0);
3759         int vhno = VFSHASH(getmajor(dev), getminor(dev));
3760         kmutex_t *hmp = &rvfs_list[vhno].rvfs_lock;

3762         mutex_enter(hmp);
3763         for (vfsp = rvfs_list[vhno].rvfs_head; vfsp; vfsp = vfsp->vfs_hash) {
3764                 if (vfsp->vfs_fsid.val[0] == val0 &&
3765                     vfsp->vfs_fsid.val[1] == val1) {
3766                         VFS_HOLD(vfsp);
3767                         mutex_exit(hmp);
3768                         return (vfsp);
3769                 }
3770         }
3771         mutex_exit(hmp);
3772         return (NULL);
3773 }

3775 /*
3776  * Search the vfs mount in progress list for a specified device/vfs entry.
3777  * Returns 0 if the first entry in the list that the device matches has the
3778  * given vfs pointer as well.  If the device matches but a different vfs
3779  * pointer is encountered in the list before the given vfs pointer then
3780  * a 1 is returned.
3781  */

3783 int
3784 vfs_devmounting(dev_t dev, struct vfs *vfsp)
3785 {
3786         int retval = 0;
3787         struct ipmnt *mipp;

3789         mutex_enter(&vfs_miplist_mutex);
3790         for (mipp = vfs_miplist; mipp != NULL; mipp = mipp->mip_next) {
3791                 if (mipp->mip_dev == dev) {
3792                         if (mipp->mip_vfsp != vfsp)
3793                                 retval = 1;
3794                         break;
3795                 }
3796         }
3797         mutex_exit(&vfs_miplist_mutex);
3798         return (retval);
3799 }

3801 /*
3802  * Search the vfs list for a specified device.  Returns 1, if entry is found
3803  * or 0 if no suitable entry is found.
3804  */

3806 int
3807 vfs_devismounted(dev_t dev)
3808 {
3809         struct vfs *vfsp;
3810         int found = 0;
3811         struct vfs search;
3812         avl_index_t index;

3814         search.vfs_dev = dev;
3815         search.vfs_mntix = 0;
  88         int found;

3817         vfs_list_read_lock();

3819         /*
3820          * there might be several entries with the same dev in the tree,
3821          * only discerned by mntix. To find the first, we start with a mntix
3822          * of 0. The search will fail. The following avl_nearest will give
```

```
3823          * us the actual first entry.
3824          */
3825         VERIFY(avl_find(&vfs_by_dev, &search, &index) == NULL);
3826         vfsp = avl_nearest(&vfs_by_dev, index, AVL_AFTER);

3828         if (vfsp != NULL && vfsp->vfs_dev == dev)
  91         vfsp = rootvfs;
  92         found = 0;
  93         do {
  94                 if (vfsp->vfs_dev == dev) {
3829                         found = 1;
  96                         break;
  97                 }
  98                 vfsp = vfsp->vfs_next;
  99         } while (vfsp != rootvfs);

3831         vfs_list_unlock();
3832         return (found);
3833 }

3835 /*
3836  * Search the vfs list for a specified device.  Returns a pointer to it
3837  * or NULL if no suitable entry is found. The caller of this routine
3838  * is responsible for releasing the returned vfs pointer.
3839  */
3840 struct vfs *
3841 vfs_dev2vfsp(dev_t dev)
3842 {
3843         struct vfs *vfsp;
3844         int found;
3845         struct vfs search;
3846         avl_index_t index;

3848         search.vfs_dev = dev;
3849         search.vfs_mntix = 0;
3850 #endif /* ! codereview */

3852         vfs_list_read_lock();

3854         /*
3855          * there might be several entries with the same dev in the tree,
3856          * only discerned by mntix. To find the first, we start with a mntix
3857          * of 0. The search will fail. The following avl_nearest will give
3858          * us the actual first entry.
3859          */
3860         VERIFY(avl_find(&vfs_by_dev, &search, &index) == NULL);
3861         vfsp = avl_nearest(&vfs_by_dev, index, AVL_AFTER);

 115         vfsp = rootvfs;
3863         found = 0;
3864         while (vfsp != NULL && vfsp->vfs_dev == dev) {
 117         do {
3865                 /*
3866                  * The following could be made more efficient by making
3867                  * the entire loop use vfs_zone_next if the call is from
3868                  * a zone.  The only callers, however, ustat(2) and
3869                  * umount2(2), don't seem to justify the added
3870                  * complexity at present.
3871                  */
3872                 if (ZONE_PATH_VISIBLE(refstr_value(vfsp->vfs_mntpt),
 125                 if (vfsp->vfs_dev == dev &&
 126                     ZONE_PATH_VISIBLE(refstr_value(vfsp->vfs_mntpt),
3873                     curproc->p_zone)) {
3874                         VFS_HOLD(vfsp);
3875                         found = 1;
3876                         break;
```



```
3877                 }
3878                 vfsp = AVL_NEXT(&vfs_by_dev, vfsp);
3879         }
 132         vfsp = vfsp->vfs_next;
 133         } while (vfsp != rootvfs);
3880         vfs_list_unlock();
3881         return (found ? vfsp : NULL);
 135         return (found ? vfsp: NULL);
3882 }

3884 /*
3885  * Search the vfs list for a specified mntpoint.  Returns a pointer to it
3886  * or NULL if no suitable entry is found. The caller of this routine
3887  * is responsible for releasing the returned vfs pointer.
3888  *
3889  * Note that if multiple mntpoints match, the last one matching is
3890  * returned in an attempt to return the "top" mount when overlay
3891  * mounts are covering the same mount point.  This is accomplished by starting
3892  * at the end of the list and working our way backwards, stopping at the first
3893  * matching mount.
3894  */
3895 struct vfs *
3896 vfs_mntpoint2vfsp(const char *mp)
3897 {
3898         struct vfs *vfsp;
3899         struct vfs *retvfsp = NULL;
3900         zone_t *zone = curproc->p_zone;
3901         struct vfs *list;

3903         vfs_list_read_lock();
3904         if (getzoneid() == GLOBAL_ZONEID) {
3905                 /*
3906                  * The global zone may see filesystems in any zone.
3907                  */
3908                 struct vfs search;
3909                 search.vfs_mntpt = refstr_alloc(mp);
3910                 search.vfs_mntix = UINT64_MAX;
3911                 avl_index_t index;

3913                 /*
3914                  * there might be several entries with the same mntpnt in the
3915                  * tree, only discerned by mntix. To find the last, we start
3916                  * with a mntix of UINT64_MAX. The search will fail. The
3917                  * following avl_nearest will give  us the actual last entry
3918                  * matching the mntpnt.
3919                  */
3920                 VERIFY(avl_find(&vfs_by_mntpnt, &search, &index) == 0);
3921                 vfsp = avl_nearest(&vfs_by_mntpnt, index, AVL_BEFORE);

3923                 refstr_rele(search.vfs_mntpt);

3925                 if (vfsp != NULL &&
3926                     strcmp(refstr_value(vfsp->vfs_mntpt), mp) == 0)
 162                 vfsp = rootvfs->vfs_prev;
 163                 do {
 164                         if (strcmp(refstr_value(vfsp->vfs_mntpt), mp) == 0) {
3927                         retvfsp = vfsp;
 166                                 break;
 167                         }
 168                         vfsp = vfsp->vfs_prev;
 169                 } while (vfsp != rootvfs->vfs_prev);
3928         } else if ((list = zone->zone_vfslist) != NULL) {
3929                 const char *mntpt;

3931                 vfsp = list->vfs_zone_prev;
3932                 do {
```

```
3933                        mntpt = refstr_value(vfsp->vfs_mntpt);
3934                        mntpt = ZONE_PATH_TRANSLATE(mntpt, zone);
3935                        if (strcmp(mntpt, mp) == 0) {
3936                                retvfsp = vfsp;
3937                                break;
3938                        }
3939                        vfsp = vfsp->vfs_zone_prev;
3940                } while (vfsp != list->vfs_zone_prev);
3941        }
3942        if (retvfsp)
3943                VFS_HOLD(retvfsp);
3944        vfs_list_unlock();
3945        return (retvfsp);
3946 }
```

_____***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_

 174 extern avl_tree_t       vskstat_tree;
 175 extern kmutex_t         vskstat_tree_lock;

 177 /*
 178  * Structure per mounted file system.  Each mounted file system has
 179  * an array of operations and an instance record.
 180  *
 181  * The file systems are kept on a doubly linked circular list headed by
 182  * "rootvfs".
 183  * File system implementations should not access this list;
 184  * it's intended for use only in the kernel's vfs layer.
 185  *
 186  * Each zone also has its own list of mounts, containing filesystems mounted
 187  * somewhere within the filesystem tree rooted at the zone's rootpath.  The
 188  * list is doubly linked to match the global list.
 189  *
 190  * mnttab locking: the in-kernel mnttab uses the vfs_mntpt, vfs_resource and
 191  * vfs_mntopts fields in the vfs_t. mntpt and resource are refstr_ts that
 192  * are set at mount time and can only be modified during a remount.
 193  * It is safe to read these fields if you can prevent a remount on the vfs,
 194  * or through the convenience funcs vfs_getmntpoint() and vfs_getresource().
 195  * The mntopts field may only be accessed through the provided convenience
 196  * functions, as it is protected by the vfs list lock. Modifying a mount
 197  * option requires grabbing the vfs list write lock, which can be a very
 198  * high latency lock.
 199  */
 200 struct zone;            /* from zone.h */
 201 struct fem_head;        /* from fem.h */

 203 typedef struct vfs {
 204         struct vfs      *vfs_next;              /* next VFS in VFS list */
 205         struct vfs      *vfs_prev;              /* prev VFS in VFS list */
 206         avl_node_t      vfs_avldev;             /* by dev index */
 207         avl_node_t      vfs_avlmntpnt;          /* by mntpnt index */
 208         /*
 209          * global mount count to define an order on entries in
 210          * the avl trees with same dev/mountpoint
 211          */
 212         uint64_t        vfs_mntix;
 213 #endif /* ! codereview */

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

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



```
 233         time_t          vfs_mtime;              /* time we were mounted */
 234         struct vfs_impl *vfs_implp;             /* impl specific data */
 235         /*
 236          * Zones support.  Note that the zone that "owns" the mount isn't
 237          * necessarily the same as the zone in which the zone is visible.
 238          * That is, vfs_zone and (vfs_zone_next|vfs_zone_prev) may refer to
 239          * different zones.
 240          */
 241         struct zone     *vfs_zone;              /* zone that owns the mount */
 242         struct vfs      *vfs_zone_next;         /* next VFS visible in zone */
 243         struct vfs      *vfs_zone_prev;         /* prev VFS visible in zone */

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

 249 #define vfs_featureset  vfs_implp->vi_featureset
 250 #define vfs_vskap       vfs_implp->vi_vskap
 251 #define vfs_fstypevsp   vfs_implp->vi_fstypevsp
 252 #define vfs_vopstats    vfs_implp->vi_vopstats
 253 #define vfs_hrctime     vfs_implp->vi_hrctime

 255 /*
 256  * VFS flags.
 257  */
 258 #define VFS_RDONLY      0x01            /* read-only vfs */
 259 #define VFS_NOMNTTAB    0x02            /* vfs not seen in mnttab */
 260 #define VFS_NOSETUID    0x08            /* setuid disallowed */
 261 #define VFS_REMOUNT     0x10            /* modify mount options only */
 262 #define VFS_NOTRUNC     0x20            /* does not truncate long file names */
 263 #define VFS_UNLINKABLE  0x40            /* unlink(2) can be applied to root */
 264 #define VFS_PXFS        0x80            /* clustering: global fs proxy vfs */
 265 #define VFS_UNMOUNTED   0x100           /* file system has been unmounted */
 266 #define VFS_NBMAND      0x200           /* allow non-blocking mandatory locks */
 267 #define VFS_XATTR       0x400           /* fs supports extended attributes */
 268 #define VFS_NODEVICES   0x800           /* device-special files disallowed */
 269 #define VFS_NOEXEC      0x1000          /* executables disallowed */
 270 #define VFS_STATS       0x2000          /* file system can collect stats */
 271 #define VFS_XID         0x4000          /* file system supports extended ids */

 273 #define VFS_NORESOURCE  "unspecified_resource"
 274 #define VFS_NOMNTPT     "unspecified_mountpoint"

 276 /*
 277  * VFS features are implemented as bits set in the vfs_t.
 278  * The vfs_feature_t typedef is a 64-bit number that will translate
 279  * into an element in an array of bitmaps and a bit in that element.
 280  * Developers must not depend on the implementation of this and
 281  * need to use vfs_has_feature()/vfs_set_feature() routines.
 282  */
 283 typedef uint64_t        vfs_feature_t;

 285 #define VFSFT_XVATTR            0x100000001     /* Supports xvattr for attrs */
 286 #define VFSFT_CASEINSENSITIVE   0x100000002     /* Supports case-insensitive */
 287 #define VFSFT_NOCASESENSITIVE   0x100000004     /* NOT case-sensitive */
 288 #define VFSFT_DIRENTFLAGS       0x100000008     /* Supports dirent flags */
 289 #define VFSFT_ACLONCREATE       0x100000010     /* Supports ACL on create */
 290 #define VFSFT_ACEMASKONACCESS   0x100000020     /* Can use ACEMASK for access */
 291 #define VFSFT_SYSATTR_VIEWS     0x100000040     /* Supports sysattr view i/f */
 292 #define VFSFT_ACCESS_FILTER     0x100000080     /* dirents filtered by access */
 293 #define VFSFT_REPARSE           0x100000100     /* Supports reparse point */
 294 #define VFSFT_ZEROCOPY_SUPPORTED        0x100000200
 295                                 /* Support loaning /returning cache buffer */
 296 /*
 297  * Argument structure for mount(2).
 298  *
```

```
 * 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);
```

```
563 int     vfs_zone_change_safe(vfs_t *);

565 int     vfs_get_lofi(vfs_t *, vnode_t **);

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

571 /*
572  * Globals.
573  */

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

581 /*
582  * The following variables are private to the the kernel's vfs layer.  File
583  * system implementations should not access them.
584  */
585 extern struct vfs *rootvfs;             /* ptr to root vfs structure */
586 typedef struct {
587         struct vfs *rvfs_head;          /* head vfs in chain */
588         kmutex_t rvfs_lock;             /* mutex protecting this chain */
589         uint32_t rvfs_len;              /* length of this chain */
590 } rvfs_t;
591 extern rvfs_t *rvfs_list;
592 extern int vfshsz;                      /* # of elements in rvfs_head array */
593 extern const mntopts_t vfs_mntopts;     /* globally recognized options */

595 #endif /* defined(_KERNEL) */

597 #define VFS_HOLD(vfsp) { \
598         vfs_hold(vfsp); \
599 }

601 #define VFS_RELE(vfsp)  { \
602         vfs_rele(vfsp); \
603 }

605 #define VFS_INIT(vfsp, op, data) { \
606         vfs_init((vfsp), (op), (data)); \
607 }


610 #define VFS_INSTALLED(vfsswp)   (((vfsswp)->vsw_flag & VSW_INSTALLED) != 0)
611 #define ALLOCATED_VFSSW(vswp)           ((vswp)->vsw_name[0] != '\0')
612 #define RLOCK_VFSSW()                   (rw_enter(&vfssw_lock, RW_READER))
613 #define RUNLOCK_VFSSW()                 (rw_exit(&vfssw_lock))
614 #define WLOCK_VFSSW()                   (rw_enter(&vfssw_lock, RW_WRITER))
615 #define WUNLOCK_VFSSW()                 (rw_exit(&vfssw_lock))
616 #define VFSSW_LOCKED()                  (RW_LOCK_HELD(&vfssw_lock))
617 #define VFSSW_WRITE_LOCKED()            (RW_WRITE_HELD(&vfssw_lock))
618 /*
619  * VFS_SYNC flags.
620  */
621 #define SYNC_ATTR       0x01            /* sync attributes only */
622 #define SYNC_CLOSE      0x02            /* close open file */
623 #define SYNC_ALL        0x04            /* force to sync all fs */

625 #ifdef  __cplusplus
626 }
627 #endif
```



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