```
**********************************************************
   17940 Tue Aug 31 13:02:02 2010
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_client.c
6879933 Let SMBFS support extensible attributes per. PSARC 2007/315
**********************************************************
```

```
/*
 * 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) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 *
 *      Copyright (c) 1983,1984,1985,1986,1987,1988,1989  AT&T.
 *      All rights reserved.
 */

#include <sys/param.h>
#include <sys/systm.h>
#include <sys/thread.h>
#include <sys/t_lock.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <sys/vfs.h>
#include <sys/errno.h>
#include <sys/buf.h>
#include <sys/stat.h>
#include <sys/cred.h>
#include <sys/kmem.h>
#include <sys/debug.h>
#include <sys/vmsystm.h>
#include <sys/flock.h>
#include <sys/share.h>
#include <sys/cmn_err.h>
#include <sys/tiuser.h>
#include <sys/sysmacros.h>
#include <sys/callb.h>
#include <sys/acl.h>
#include <sys/kstat.h>
#include <sys/signal.h>
#include <sys/list.h>
#include <sys/zone.h>

#include <netsmb/smb.h>
#include <netsmb/smb_conn.h>
#include <netsmb/smb_subr.h>

#include <smbfs/smbfs.h>
#include <smbfs/smbfs_node.h>
#include <smbfs/smbfs_subr.h>
```



```
#include <vm/hat.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/seg.h>
#include <vm/seg_map.h>
#include <vm/seg_vn.h>

static int smbfs_getattr_cache(vnode_t *, smbfattr_t *);
static void smbfattr_to_vattr(vnode_t *, smbfattr_t *, vattr_t *);
static void smbfattr_to_xvattr(vnode_t *, smbfattr_t *, vattr_t *);
static int smbfs_getattr_cache(vnode_t *, struct smbfattr *);
static int smbfattr_to_vattr(vnode_t *, struct smbfattr *,
        struct vattr *);

/*
 * The following code provide zone support in order to perform an action
 * for each smbfs mount in a zone.  This is also where we would add
 * per-zone globals and kernel threads for the smbfs module (since
 * they must be terminated by the shutdown callback).
 */

struct smi_globals {
        kmutex_t        smg_lock;  /* lock protecting smg_list */
        list_t          smg_list;  /* list of SMBFS mounts in zone */
        boolean_t       smg_destructor_called;
};
_____unchanged_portion_omitted_

/*
 * Return either cached or remote attributes. If get remote attr
 * use them to check and invalidate caches, then cache the new attributes.
 *
 * From NFS: nfsgetattr()
 */
int
smbfsgetattr(vnode_t *vp, struct vattr *vap, cred_t *cr)
{
        struct smbfattr fa;
        smbmntinfo_t *smi;
        uint_t mask;
        int error;

        smi = VTOSMI(vp);

        ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);

        /*
         * If asked for UID or GID, update n_uid, n_gid.
         */
        mask = AT_ALL;
        if (vap->va_mask & (AT_UID | AT_GID)) {
                if (smi->smi_flags & SMI_ACL)
                        (void) smbfs_acl_getids(vp, cr);
                /* else leave as set in make_smbnode */
        } else {
                mask &= ~(AT_UID | AT_GID);
        }

        /*
         * If we've got cached attributes, just use them;
         * otherwise go to the server to get attributes,
         * which will update the cache in the process.
         */
        error = smbfs_getattr_cache(vp, &fa);
```

```
        if (error)
                error = smbfs_getattr_otw(vp, &fa, cr);
        if (error)
                return (error);
        vap->va_mask |= mask;

        /*
         * Re. client's view of the file size, see:
         * smbfs_attrcache_fa, smbfs_getattr_otw
         */
        smbfattr_to_vattr(vp, &fa, vap);
        if (vap->va_mask & AT_XVATTR)
                smbfattr_to_xvattr(vp, &fa, vap);

        return (0);
        error = smbfattr_to_vattr(vp, &fa, vap);
        vap->va_mask = mask;

        return (error);
}


/*
 * Convert SMB over the wire attributes to vnode form.
 * Returns 0 for success, error if failed (overflow, etc).
 * From NFS: nattr_to_vattr()
 */
void
int
smbfattr_to_vattr(vnode_t *vp, struct smbfattr *fa, struct vattr *vap)
{
        struct smbnode *np = VTOSMB(vp);

        /* Set va_mask in caller */

        /*
         * Take type, mode, uid, gid from the smbfs node,
         * which has have been updated by _getattr_otw.
         */
        vap->va_type = vp->v_type;
        vap->va_mode = np->n_mode;

        vap->va_uid = np->n_uid;
        vap->va_gid = np->n_gid;

        vap->va_fsid = vp->v_vfsp->vfs_dev;
        vap->va_nodeid = np->n_ino;
        vap->va_nlink = 1;

        /*
         * Difference from NFS here:  We cache attributes as
         * reported by the server, so r_attr.fa_size is the
         * server's idea of the file size.  This is called
         * for getattr, so we want to return the client's
         * idea of the file size.  NFS deals with that in
         * nfsgetattr(), the equivalent of our caller.
         */
        vap->va_size = np->r_size;

        /*
         * Times.  Note, already converted from NT to
         * Unix form (in the unmarshalling code).
         */
        vap->va_atime = fa->fa_atime;
        vap->va_mtime = fa->fa_mtime;
        vap->va_ctime = fa->fa_ctime;
```



```
        /*
         * rdev, blksize, seq are made up.
         * va_nblocks is 512 byte blocks.
         */
        vap->va_rdev = vp->v_rdev;
        vap->va_blksize = MAXBSIZE;
        vap->va_nblocks = (fsblkcnt64_t)btod(np->r_attr.fa_allocsz);
        vap->va_seq = 0;
}

/*
 * smbfattr_to_xvattr: like smbfattr_to_vattr but for
 * Extensible system attributes (PSARC 2007/315)
 */
static void
smbfattr_to_xvattr(vnode_t *vp, struct smbfattr *fa, struct vattr *vap)
{
        struct smbnode *np = VTOSMB(vp);
        xvattr_t *xvap = (xvattr_t *)vap;       /* *vap may be xvattr_t */
        xoptattr_t *xoap = NULL;

        if ((xoap = xva_getxoptattr(xvap)) == NULL)
                return;

        if (XVA_ISSET_REQ(xvap, XAT_CREATETIME)) {
                xoap->xoa_createtime = fa->fa_createtime;
                XVA_SET_RTN(xvap, XAT_CREATETIME);
        }

        if (XVA_ISSET_REQ(xvap, XAT_ARCHIVE)) {
                xoap->xoa_archive =
                    ((fa->fa_attr & SMB_FA_ARCHIVE) != 0);
                XVA_SET_RTN(xvap, XAT_ARCHIVE);
        }

        if (XVA_ISSET_REQ(xvap, XAT_SYSTEM)) {
                xoap->xoa_system =
                    ((fa->fa_attr & SMB_FA_SYSTEM) != 0);
                XVA_SET_RTN(xvap, XAT_SYSTEM);
        }

        if (XVA_ISSET_REQ(xvap, XAT_READONLY)) {
                xoap->xoa_readonly =
                    ((fa->fa_attr & SMB_FA_RDONLY) != 0);
                XVA_SET_RTN(xvap, XAT_READONLY);
        }

        if (XVA_ISSET_REQ(xvap, XAT_HIDDEN)) {
                xoap->xoa_hidden =
                    ((fa->fa_attr & SMB_FA_HIDDEN) != 0);
                XVA_SET_RTN(xvap, XAT_HIDDEN);
        }
        return (0);
}


/*
 * SMB Client initialization and cleanup.
 * Much of it is per-zone now.
 */


/* ARGSUSED */
static void *
smbfs_zone_init(zoneid_t zoneid)
```

```
 563 {
 564         smi_globals_t *smg;

 566         smg = kmem_alloc(sizeof (*smg), KM_SLEEP);
 567         mutex_init(&smg->smg_lock, NULL, MUTEX_DEFAULT, NULL);
 568         list_create(&smg->smg_list, sizeof (smbmntinfo_t),
 569             offsetof(smbmntinfo_t, smi_zone_node));
 570         smg->smg_destructor_called = B_FALSE;
 571         return (smg);
 572 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   24755 Tue Aug 31 13:02:03 2010
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vfsops.c
6879933 Let SMBFS support extensible attributes per. PSARC 2007/315
**********************************************************
______unchanged_portion_omitted_

 331 /*
 332  * smbfs mount vfsop
 333  * Set up mount info record and attach it to vfs struct.
 334  */
 335 static int
 336 smbfs_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
 337 {
 338         char            *data = uap->dataptr;
 339         int             error;
 340         smbnode_t       *rtnp = NULL;   /* root of this fs */
 341         smbmntinfo_t    *smi = NULL;
 342         dev_t           smbfs_dev;
 343         int             version;
 344         int             devfd;
 345         zone_t          *zone = curproc->p_zone;
 346         zone_t          *mntzone = NULL;
 347         smb_share_t     *ssp = NULL;
 348         smb_cred_t      scred;
 349         int             flags, sec;

 351         STRUCT_DECL(smbfs_args, args);          /* smbfs mount arguments */

 353         if ((error = secpolicy_fs_mount(cr, mvp, vfsp)) != 0)
 354                 return (error);

 356         if (mvp->v_type != VDIR)
 357                 return (ENOTDIR);

 359         /*
 360          * get arguments
 361          *
 362          * uap->datalen might be different from sizeof (args)
 363          * in a compatible situation.
 364          */
 365         STRUCT_INIT(args, get_udatamodel());
 366         bzero(STRUCT_BUF(args), SIZEOF_STRUCT(smbfs_args, DATAMODEL_NATIVE));
 367         if (copyin(data, STRUCT_BUF(args), MIN(uap->datalen,
 368             SIZEOF_STRUCT(smbfs_args, DATAMODEL_NATIVE))))
 369                 return (EFAULT);

 371         /*
 372          * Check mount program version
 373          */
 374         version = STRUCT_FGET(args, version);
 375         if (version != SMBFS_VERSION) {
 376                 cmn_err(CE_WARN, "mount version mismatch:"
 377                     " kernel=%d, mount=%d\n",
 378                     SMBFS_VERSION, version);
 379                 return (EINVAL);
 380         }

 382         /*
 383          * Deal with re-mount requests.
 384          */
 385         if (uap->flags & MS_REMOUNT) {
 386                 cmn_err(CE_WARN, "MS_REMOUNT not implemented");
 387                 return (ENOTSUP);
 388         }
```



```
 390         /*
 391          * Check for busy
 392          */
 393         mutex_enter(&mvp->v_lock);
 394         if (!(uap->flags & MS_OVERLAY) &&
 395             (mvp->v_count != 1 || (mvp->v_flag & VROOT))) {
 396                 mutex_exit(&mvp->v_lock);
 397                 return (EBUSY);
 398         }
 399         mutex_exit(&mvp->v_lock);

 401         /*
 402          * Get the "share" from the netsmb driver (ssp).
 403          * It is returned with a "ref" (hold) for us.
 404          * Release this hold: at errout below, or in
 405          * smbfs_freevfs().
 406          */
 407         devfd = STRUCT_FGET(args, devfd);
 408         error = smb_dev2share(devfd, &ssp);
 409         if (error) {
 410                 cmn_err(CE_WARN, "invalid device handle %d (%d)\n",
 411                     devfd, error);
 412                 return (error);
 413         }

 415         /*
 416          * Use "goto errout" from here on.
 417          * See: ssp, smi, rtnp, mntzone
 418          */

 420         /*
 421          * Determine the zone we're being mounted into.
 422          */
 423         zone_hold(mntzone = zone);              /* start with this assumption */
 424         if (getzoneid() == GLOBAL_ZONEID) {
 425                 zone_rele(mntzone);
 426                 mntzone = zone_find_by_path(refstr_value(vfsp->vfs_mntpt));
 427                 ASSERT(mntzone != NULL);
 428                 if (mntzone != zone) {
 429                         error = EBUSY;
 430                         goto errout;
 431                 }
 432         }

 434         /*
 435          * Stop the mount from going any further if the zone is going away.
 436          */
 437         if (zone_status_get(mntzone) >= ZONE_IS_SHUTTING_DOWN) {
 438                 error = EBUSY;
 439                 goto errout;
 440         }

 442         /*
 443          * On a Trusted Extensions client, we may have to force read-only
 444          * for read-down mounts.
 445          */
 446         if (is_system_labeled()) {
 447                 void *addr;
 448                 int ipvers = 0;
 449                 struct smb_vc *vcp;

 451                 vcp = SSTOVC(ssp);
 452                 addr = smb_vc_getipaddr(vcp, &ipvers);
 453                 error = smbfs_mount_label_policy(vfsp, addr, ipvers, cr);

 455                 if (error > 0)
```

```
456                         goto errout;
458                 if (error == -1) {
459                         /* change mount to read-only to prevent write-down */
460                         vfs_setmntopt(vfsp, MNTOPT_RO, NULL, 0);
461                 }
462         }
464         /* Prevent unload. */
465         atomic_inc_32(&smbfs_mountcount);
467         /*
468          * Create a mount record and link it to the vfs struct.
469          * No more possiblities for errors from here on.
470          * Tear-down of this stuff is in smbfs_free_smi()
471          *
472          * Compare with NFS: nfsrootvp()
473          */
474         smi = kmem_zalloc(sizeof (*smi), KM_SLEEP);
476         mutex_init(&smi->smi_lock, NULL, MUTEX_DEFAULT, NULL);
477         cv_init(&smi->smi_statvfs_cv, NULL, CV_DEFAULT, NULL);
479         rw_init(&smi->smi_hash_lk, NULL, RW_DEFAULT, NULL);
480         smbfs_init_hash_avl(&smi->smi_hash_avl);
482         smi->smi_share = ssp;
483         ssp = NULL;
485         /*
486          * Convert the anonymous zone hold acquired via zone_hold() above
487          * into a zone reference.
488          */
489         zone_init_ref(&smi->smi_zone_ref);
490         zone_hold_ref(mntzone, &smi->smi_zone_ref, ZONE_REF_SMBFS);
491         zone_rele(mntzone);
492         mntzone = NULL;
494         /*
495          * Initialize option defaults
496          */
497         smi->smi_flags  = SMI_LLOCK;
498         smi->smi_acregmin = SEC2HR(SMBFS_ACREGMIN);
499         smi->smi_acregmax = SEC2HR(SMBFS_ACREGMAX);
500         smi->smi_acdirmin = SEC2HR(SMBFS_ACDIRMIN);
501         smi->smi_acdirmax = SEC2HR(SMBFS_ACDIRMAX);
503         /*
504          * All "generic" mount options have already been
505          * handled in vfs.c:domount() - see mntopts stuff.
506          * Query generic options using vfs_optionisset().
507          */
508         if (vfs_optionisset(vfsp, MNTOPT_INTR, NULL))
509                 smi->smi_flags |= SMI_INT;
510         if (vfs_optionisset(vfsp, MNTOPT_ACL, NULL))
511                 smi->smi_flags |= SMI_ACL;
513         /*
514          * Get the mount options that come in as smbfs_args,
515          * starting with args.flags (SMBFS_MF_xxx)
516          */
517         flags = STRUCT_FGET(args, flags);
518         smi->smi_uid    = STRUCT_FGET(args, uid);
519         smi->smi_gid    = STRUCT_FGET(args, gid);
520         smi->smi_fmode  = STRUCT_FGET(args, file_mode) & 0777;
521         smi->smi_dmode  = STRUCT_FGET(args, dir_mode) & 0777;
```



```
523         /*
524          * Hande the SMBFS_MF_xxx flags.
525          */
526         if (flags & SMBFS_MF_NOAC)
527                 smi->smi_flags |= SMI_NOAC;
528         if (flags & SMBFS_MF_ACREGMIN) {
529                 sec = STRUCT_FGET(args, acregmin);
530                 if (sec < 0 || sec > SMBFS_ACMINMAX)
531                         sec = SMBFS_ACMINMAX;
532                 smi->smi_acregmin = SEC2HR(sec);
533         }
534         if (flags & SMBFS_MF_ACREGMAX) {
535                 sec = STRUCT_FGET(args, acregmax);
536                 if (sec < 0 || sec > SMBFS_ACMAXMAX)
537                         sec = SMBFS_ACMAXMAX;
538                 smi->smi_acregmax = SEC2HR(sec);
539         }
540         if (flags & SMBFS_MF_ACDIRMIN) {
541                 sec = STRUCT_FGET(args, acdirmin);
542                 if (sec < 0 || sec > SMBFS_ACMINMAX)
543                         sec = SMBFS_ACMINMAX;
544                 smi->smi_acdirmin = SEC2HR(sec);
545         }
546         if (flags & SMBFS_MF_ACDIRMAX) {
547                 sec = STRUCT_FGET(args, acdirmax);
548                 if (sec < 0 || sec > SMBFS_ACMAXMAX)
549                         sec = SMBFS_ACMAXMAX;
550                 smi->smi_acdirmax = SEC2HR(sec);
551         }
553         /*
554          * Get attributes of the remote file system,
555          * i.e. ACL support, named streams, etc.
556          */
557         smb_credinit(&scred, cr);
558         error = smbfs_smb_qfsattr(smi->smi_share, &smi->smi_fsa, &scred);
559         smb_credrele(&scred);
560         if (error) {
561                 SMBVDEBUG("smbfs_smb_qfsattr error %d\n", error);
562         }
564         /*
565          * We enable XATTR by default (via smbfs_mntopts)
566          * but if the share does not support named streams,
567          * force the NOXATTR option (also clears XATTR).
568          * Caller will set or clear VFS_XATTR after this.
569          */
570         if ((smi->smi_fsattr & FILE_NAMED_STREAMS) == 0)
571                 vfs_setmntopt(vfsp, MNTOPT_NOXATTR, NULL, 0);
573         /*
574          * Ditto ACLs (disable if not supported on this share)
575          */
576         if ((smi->smi_fsattr & FILE_PERSISTENT_ACLS) == 0) {
577                 vfs_setmntopt(vfsp, MNTOPT_NOACL, NULL, 0);
578                 smi->smi_flags &= ~SMI_ACL;
579         }
581         /*
582          * Assign a unique device id to the mount
583          */
584         mutex_enter(&smbfs_minor_lock);
585         do {
586                 smbfs_minor = (smbfs_minor + 1) & MAXMIN32;
587                 smbfs_dev = makedevice(smbfs_major, smbfs_minor);
```

```
588         } while (vfs_devismounted(smbfs_dev));
589         mutex_exit(&smbfs_minor_lock);

591         vfsp->vfs_dev   = smbfs_dev;
592         vfs_make_fsid(&vfsp->vfs_fsid, smbfs_dev, smbfsfstyp);
593         vfsp->vfs_data  = (caddr_t)smi;
594         vfsp->vfs_fstype = smbfsfstyp;
595         vfsp->vfs_bsize = MAXBSIZE;
596         vfsp->vfs_bcount = 0;

598         smi->smi_vfsp   = vfsp;
599         smbfs_zonelist_add(smi);        /* undo in smbfs_freevfs */

601         /* PSARC 2007/227 VFS Feature Registration */
602         vfs_set_feature(vfsp, VFSFT_XVATTR);
603         vfs_set_feature(vfsp, VFSFT_SYSATTR_VIEWS);

605         /*
606          * Create the root vnode, which we need in unmount
607          * for the call to smbfs_check_table(), etc.
608          * Release this hold in smbfs_unmount.
609          */
610         rtnp = smbfs_node_findcreate(smi, "\\", 1, NULL, 0, 0,
611             &smbfs_fattr0);
612         ASSERT(rtnp != NULL);
613         rtnp->r_vnode->v_type = VDIR;
614         rtnp->r_vnode->v_flag |= VROOT;
615         smi->smi_root = rtnp;

617         /*
618          * NFS does other stuff here too:
619          *   async worker threads
620          *   init kstats
621          *
622          * End of code from NFS nfsrootvp()
623          */
624         return (0);

626 errout:
627         vfsp->vfs_data = NULL;
628         if (smi != NULL)
629                 smbfs_free_smi(smi);

631         if (mntzone != NULL)
632                 zone_rele(mntzone);

634         if (ssp != NULL)
635                 smb_share_rele(ssp);

637         return (error);
638 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   78744 Tue Aug 31 13:02:04 2010
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c
6879933 Let SMBFS support extensible attributes per. PSARC 2007/315
**********************************************************
______unchanged_portion_omitted_

  97 /*
  98  * Turning this on causes nodes to be created in the cache
  99  * during directory listings, normally avoiding a second
 100  * OtW attribute fetch just after a readdir.
 101  */
 102 int smbfs_fastlookup = 1;

 104 /* local static function defines */

 106 static int      smbfslookup_cache(vnode_t *, char *, int, vnode_t **,
 107                         cred_t *);
 108 static int      smbfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
 109                         int cache_ok, caller_context_t *);
 110 static int      smbfsrename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm,
 111                         cred_t *cr, caller_context_t *);
 112 static int      smbfssetattr(vnode_t *, struct vattr *, int, cred_t *);
 113 static int      smbfs_accessx(void *, int, cred_t *);
 114 static int      smbfs_readvdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
 115                         caller_context_t *);
 116 static void     smbfs_rele_fid(smbnode_t *, struct smb_cred *);
 117 static uint32_t xvattr_to_dosattr(smbnode_t *, struct vattr *);

 119 /*
 120  * These are the vnode ops routines which implement the vnode interface to
 121  * the networked file system.  These routines just take their parameters,
 122  * make them look networkish by putting the right info into interface structs,
 123  * and then calling the appropriate remote routine(s) to do the work.
 124  *
 125  * Note on directory name lookup cacheing:  If we detect a stale fhandle,
 126  * we purge the directory cache relative to that vnode.  This way, the
 127  * user won't get burned by the cache repeatedly.  See <smbfs/smbnode.h> for
 128  * more details on smbnode locking.
 129  */

 131 static int      smbfs_open(vnode_t **, int, cred_t *, caller_context_t *);
 132 static int      smbfs_close(vnode_t *, int, int, offset_t, cred_t *,
 133                         caller_context_t *);
 134 static int      smbfs_read(vnode_t *, struct uio *, int, cred_t *,
 135                         caller_context_t *);
 136 static int      smbfs_write(vnode_t *, struct uio *, int, cred_t *,
 137                         caller_context_t *);
 138 static int      smbfs_ioctl(vnode_t *, int, intptr_t, int, cred_t *, int *,
 139                         caller_context_t *);
 140 static int      smbfs_getattr(vnode_t *, struct vattr *, int, cred_t *,
 141                         caller_context_t *);
 142 static int      smbfs_setattr(vnode_t *, struct vattr *, int, cred_t *,
 143                         caller_context_t *);
 144 static int      smbfs_access(vnode_t *, int, int, cred_t *, caller_context_t *);
 145 static int      smbfs_fsync(vnode_t *, int, cred_t *, caller_context_t *);
 146 static void     smbfs_inactive(vnode_t *, cred_t *, caller_context_t *);
 147 static int      smbfs_lookup(vnode_t *, char *, vnode_t **, struct pathname *,
 148                         int, vnode_t *, cred_t *, caller_context_t *,
 149                         int *, pathname_t *);
 150 static int      smbfs_create(vnode_t *, char *, struct vattr *, enum vcexcl,
 151                         int, vnode_t **, cred_t *, int, caller_context_t *,
 152                         vsecattr_t *);
 153 static int      smbfs_remove(vnode_t *, char *, cred_t *, caller_context_t *,
 154                         int);
 155 static int      smbfs_rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
```



```
 156                         caller_context_t *, int);
 157 static int      smbfs_mkdir(vnode_t *, char *, struct vattr *, vnode_t **,
 158                         cred_t *, caller_context_t *, int, vsecattr_t *);
 159 static int      smbfs_rmdir(vnode_t *, char *, vnode_t *, cred_t *,
 160                         caller_context_t *, int);
 161 static int      smbfs_readdir(vnode_t *, struct uio *, cred_t *, int *,
 162                         caller_context_t *, int);
 163 static int      smbfs_rwlock(vnode_t *, int, caller_context_t *);
 164 static void     smbfs_rwunlock(vnode_t *, int, caller_context_t *);
 165 static int      smbfs_seek(vnode_t *, offset_t, offset_t *, caller_context_t *);
 166 static int      smbfs_frlock(vnode_t *, int, struct flock64 *, int, offset_t,
 167                         struct flk_callback *, cred_t *, caller_context_t *);
 168 static int      smbfs_space(vnode_t *, int, struct flock64 *, int, offset_t,
 169                         cred_t *, caller_context_t *);
 170 static int      smbfs_pathconf(vnode_t *, int, ulong_t *, cred_t *,
 171                         caller_context_t *);
 172 static int      smbfs_setsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
 173                         caller_context_t *);
 174 static int      smbfs_getsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
 175                         caller_context_t *);
 176 static int      smbfs_shrlock(vnode_t *, int, struct shrlock *, int, cred_t *,
 177                         caller_context_t *);

 179 /* Dummy function to use until correct function is ported in */
 180 int noop_vnodeop() {
 181         return (0);
 182 }
______unchanged_portion_omitted_


 839 /*
 840  * Return either cached or remote attributes. If get remote attr
 841  * use them to check and invalidate caches, then cache the new attributes.
 841  *
 842  * XXX
 843  * This op should eventually support PSARC 2007/315, Extensible Attribute
 844  * Interfaces, for richer metadata.
 842  */
 843 /* ARGSUSED */
 844 static int
 845 smbfs_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
 846         caller_context_t *ct)
 847 {
 848         smbnode_t *np;
 849         smbmntinfo_t *smi;

 851         smi = VTOSMI(vp);

 853         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
 854                 return (EIO);

 856         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
 857                 return (EIO);

 859         /*
 860          * If it has been specified that the return value will
 861          * just be used as a hint, and we are only being asked
 862          * for size, fsid or rdevid, then return the client's
 863          * notion of these values without checking to make sure
 864          * that the attribute cache is up to date.
 865          * The whole point is to avoid an over the wire GETATTR
 866          * call.
 867          */
 868         np = VTOSMB(vp);
 869         if (flags & ATTR_HINT) {
 870                 if (vap->va_mask ==
```

```
871                     (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
872                         mutex_enter(&np->r_statelock);
873                         if (vap->va_mask | AT_SIZE)
874                                 vap->va_size = np->r_size;
875                         if (vap->va_mask | AT_FSID)
876                                 vap->va_fsid = vp->v_vfsp->vfs_dev;
877                         if (vap->va_mask | AT_RDEV)
878                                 vap->va_rdev = vp->v_rdev;
879                         mutex_exit(&np->r_statelock);
880                         return (0);
881                 }
882         }

884         return (smbfsgetattr(vp, vap, cr));
885 }

887 /* smbfsgetattr() in smbfs_client.c */

892 /*
893  * XXX
894  * This op should eventually support PSARC 2007/315, Extensible Attribute
895  * Interfaces, for richer metadata.
896  */
889 /*ARGSUSED4*/
890 static int
891 smbfs_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
892                 caller_context_t *ct)
893 {
894         vfs_t           *vfsp;
895         smbmntinfo_t    *smi;
896         int             error;
897         uint_t          mask;
898         struct vattr    oldva;

900         vfsp = vp->v_vfsp;
901         smi = VFTOSMI(vfsp);

903         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
904                 return (EIO);

906         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
907                 return (EIO);

909         mask = vap->va_mask;
910         if (mask & AT_NOSET)
911                 return (EINVAL);

913         if (vfsp->vfs_flag & VFS_RDONLY)
914                 return (EROFS);

916         /*
917          * This is a _local_ access check so that only the owner of
918          * this mount can set attributes.  With ACLs enabled, the
919          * file owner can be different from the mount owner, and we
920          * need to check the _mount_ owner here.  See _access_rwx
921          */
922         bzero(&oldva, sizeof (oldva));
923         oldva.va_mask = AT_TYPE | AT_MODE;
924         error = smbfsgetattr(vp, &oldva, cr);
925         if (error)
926                 return (error);
927         oldva.va_mask |= AT_UID | AT_GID;
928         oldva.va_uid = smi->smi_uid;
929         oldva.va_gid = smi->smi_gid;

931         error = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
```



```
932             smbfs_accessx, vp);
933         if (error)
934                 return (error);

936         if (mask & (AT_UID | AT_GID)) {
937                 if (smi->smi_flags & SMI_ACL)
938                         error = smbfs_acl_setids(vp, vap, cr);
939                 else
940                         error = ENOSYS;
941                 if (error != 0) {
942                         SMBVDEBUG("error %d seting UID/GID on %s",
943                             error, VTOSMB(vp)->n_rpath);
944                         /*
945                          * It might be more correct to return the
946                          * error here, but that causes complaints
947                          * when root extracts a cpio archive, etc.
948                          * So ignore this error, and go ahead with
949                          * the rest of the setattr work.
950                          */
951                 }
952         }

954         return (smbfssetattr(vp, vap, flags, cr));
955 }

957 /*
958  * Mostly from Darwin smbfs_setattr()
959  * but then modified a lot.
960  */
961 /* ARGSUSED */
962 static int
963 smbfssetattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr)
964 {
965         int             error = 0;
966         smbnode_t       *np = VTOSMB(vp);
967         uint_t          mask = vap->va_mask;
968         struct timespec *mtime, *atime;
969         struct smb_cred scred;
970         int             cerror, modified = 0;
971         unsigned short  fid;
972         int have_fid = 0;
973         uint32_t rights = 0;
974         uint32_t dosattr = 0;

976         ASSERT(curproc->p_zone == VTOSMI(vp)->smi_zone_ref.zref_zone);

978         /*
979          * There are no settable attributes on the XATTR dir,
980          * so just silently ignore these.  On XATTR files,
981          * you can set the size but nothing else.
982          */
983         if (vp->v_flag & V_XATTRDIR)
984                 return (0);
985         if (np->n_flag & N_XATTR) {
986                 if (mask & AT_TIMES)
987                         SMBVDEBUG("ignore set time on xattr\n");
988                 mask &= AT_SIZE;
989         }

991         /*
992          * If our caller is trying to set multiple attributes, they
993          * can make no assumption about what order they are done in.
994          * Here we try to do them in order of decreasing likelihood
995          * of failure, just to minimize the chance we'll wind up
996          * with a partially complete request.
997          */
```

```
 999         /* Shared lock for (possible) n_fid use. */
1000         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
1001                 return (EINTR);
1002         smb_credinit(&scred, cr);

1004         /*
1005          * If the caller has provided extensible attributes,
1006          * map those into DOS attributes supported by SMB.
1007          * Note: zero means "no change".
1008          */
1009         if (mask & AT_XVATTR)
1010                 dosattr = xvattr_to_dosattr(np, vap);

1012         /*
1013          * Will we need an open handle for this setattr?
1014          * If so, what rights will we need?
1015          */
1016         if (dosattr || (mask & (AT_ATIME | AT_MTIME))) {
1015         if (mask & (AT_ATIME | AT_MTIME)) {
1017                 rights |=
1018                     SA_RIGHT_FILE_WRITE_ATTRIBUTES;
1019         }
1020         if (mask & AT_SIZE) {
1021                 rights |=
1022                     SA_RIGHT_FILE_WRITE_DATA |
1023                     SA_RIGHT_FILE_APPEND_DATA;
1024         }

1026         /*
1027          * Only SIZE really requires a handle, but it's
1028          * simpler and more reliable to set via a handle.
1029          * Some servers like NT4 won't set times by path.
1030          * Also, we're usually setting everything anyway.
1031          */
1032         if (rights != 0) {
1031         if (mask & (AT_SIZE | AT_ATIME | AT_MTIME)) {
1033                 error = smbfs_smb_tmpopen(np, rights, &scred, &fid);
1034                 if (error) {
1035                         SMBVDEBUG("error %d opening %s\n",
1036                             error, np->n_rpath);
1037                         goto out;
1038                 }
1039                 have_fid = 1;
1040         }

1042         /*
1043          * If the server supports the UNIX extensions, right here is where
1044          * we'd support changes to uid, gid, mode, and possibly va_flags.
1045          * For now we claim to have made any such changes.
1046          */

1048         if (mask & AT_SIZE) {
1049                 /*
1050                  * If the new file size is less than what the client sees as
1051                  * the file size, then just change the size and invalidate
1052                  * the pages.
1053                  * I am commenting this code at present because the function
1054                  * smbfs_putapage() is not yet implemented.
1055                  */

1057                 /*
1058                  * Set the file size to vap->va_size.
1059                  */
1060                 ASSERT(have_fid);
1061                 error = smbfs_smb_setfsize(np, fid, vap->va_size, &scred);
```



```
1062                 if (error) {
1063                         SMBVDEBUG("setsize error %d file %s\n",
1064                             error, np->n_rpath);
1065                 } else {
1066                         /*
1067                          * Darwin had code here to zero-extend.
1068                          * Tests indicate the server will zero-fill,
1069                          * so looks like we don't need to do this.
1070                          * Good thing, as this could take forever.
1071                          *
1072                          * XXX: Reportedly, writing one byte of zero
1073                          * at the end offset avoids problems here.
1074                          */
1075                         mutex_enter(&np->r_statelock);
1076                         np->r_size = vap->va_size;
1077                         mutex_exit(&np->r_statelock);
1078                         modified = 1;
1079                 }
1080         }

1082         /*
1083          * XXX: When Solaris has create_time, set that too.
1084          * Note: create_time is different from ctime.
1085          */
1086         mtime = ((mask & AT_MTIME) ? &vap->va_mtime : 0);
1087         atime = ((mask & AT_ATIME) ? &vap->va_atime : 0);

1089         if (dosattr || mtime || atime) {
1088         if (mtime || atime) {
1090                 /*
1091                  * Always use the handle-based set attr call now.
1092                  * Not trying to set DOS attributes here so pass zero.
1093                  */
1094                 ASSERT(have_fid);
1095                 error = smbfs_smb_setfattr(np, fid,
1096                     dosattr, mtime, atime, &scred);
1095                     0, mtime, atime, &scred);
1097                 if (error) {
1098                         SMBVDEBUG("set times error %d file %s\n",
1099                             error, np->n_rpath);
1100                 } else {
1101                         modified = 1;
1102                 }
1103         }

1105 out:
1106         if (modified) {
1107                 /*
1108                  * Invalidate attribute cache in case the server
1109                  * doesn't set exactly the attributes we asked.
1110                  */
1111                 smbfs_attrcache_remove(np);
1112         }

1114         if (have_fid) {
1115                 cerror = smbfs_smb_tmpclose(np, fid, &scred);
1116                 if (cerror)
1117                         SMBVDEBUG("error %d closing %s\n",
1118                             cerror, np->n_rpath);
1119         }

1121         smb_credrele(&scred);
1122         smbfs_rw_exit(&np->r_lkserlock);

1124         return (error);
1125 }
```

```
1127 /*
1128  * Helper function for extensible system attributes (PSARC 2007/315)
1129  * Compute the DOS attribute word to pass to _setfattr (see above).
1130  * This returns zero IFF no change is being made to attributes.
1131  * Otherwise return the new attributes or SMB_EFA_NORMAL.
1132  */
1133 static uint32_t
1134 xvattr_to_dosattr(smbnode_t *np, struct vattr *vap)
1135 {
1136         xvattr_t *xvap = (xvattr_t *)vap;
1137         xoptattr_t *xoap = NULL;
1138         uint32_t attr = np->r_attr.fa_attr;
1139         boolean_t anyset = B_FALSE;
1140
1141         if ((xoap = xva_getxoptattr(xvap)) == NULL)
1142                 return (0);
1143
1144         if (XVA_ISSET_REQ(xvap, XAT_ARCHIVE)) {
1145                 if (xoap->xoa_archive)
1146                         attr |= SMB_FA_ARCHIVE;
1147                 else
1148                         attr &= ~SMB_FA_ARCHIVE;
1149                 XVA_SET_RTN(xvap, XAT_ARCHIVE);
1150                 anyset = B_TRUE;
1151         }
1152         if (XVA_ISSET_REQ(xvap, XAT_SYSTEM)) {
1153                 if (xoap->xoa_system)
1154                         attr |= SMB_FA_SYSTEM;
1155                 else
1156                         attr &= ~SMB_FA_SYSTEM;
1157                 XVA_SET_RTN(xvap, XAT_SYSTEM);
1158                 anyset = B_TRUE;
1159         }
1160         if (XVA_ISSET_REQ(xvap, XAT_READONLY)) {
1161                 if (xoap->xoa_readonly)
1162                         attr |= SMB_FA_RDONLY;
1163                 else
1164                         attr &= ~SMB_FA_RDONLY;
1165                 XVA_SET_RTN(xvap, XAT_READONLY);
1166                 anyset = B_TRUE;
1167         }
1168         if (XVA_ISSET_REQ(xvap, XAT_HIDDEN)) {
1169                 if (xoap->xoa_hidden)
1170                         attr |= SMB_FA_HIDDEN;
1171                 else
1172                         attr &= ~SMB_FA_HIDDEN;
1173                 XVA_SET_RTN(xvap, XAT_HIDDEN);
1174                 anyset = B_TRUE;
1175         }
1176
1177         if (anyset == B_FALSE)
1178                 return (0);     /* no change */
1179         if (attr == 0)
1180                 attr = SMB_EFA_NORMAL;
1181
1182         return (attr);
1183 }
1184
1185 /*
1186  * smbfs_access_rwx()
1187  * Common function for smbfs_access, etc.
1188  *
1189  * The security model implemented by the FS is unusual
1190  * due to the current "single user mounts" restriction:
1191  * All access under a given mount point uses the CIFS
```



```
1192  * credentials established by the owner of the mount.
1193  *
1194  * Most access checking is handled by the CIFS server,
1195  * but we need sufficient Unix access checks here to
1196  * prevent other local Unix users from having access
1197  * to objects under this mount that the uid/gid/mode
1198  * settings in the mount would not allow.
1199  *
1200  * With this model, there is a case where we need the
1201  * ability to do an access check before we have the
1202  * vnode for an object.  This function takes advantage
1203  * of the fact that the uid/gid/mode is per mount, and
1204  * avoids the need for a vnode.
1205  *
1206  * We still (sort of) need a vnode when we call
1207  * secpolicy_vnode_access, but that only uses
1208  * the vtype field, so we can use a pair of fake
1209  * vnodes that have only v_type filled in.
1210  *
1211  * XXX: Later, add a new secpolicy_vtype_access()
1212  * that takes the vtype instead of a vnode, and
1213  * get rid of the tmpl_vxxx fake vnodes below.
1214  */
1215 static int
1216 smbfs_access_rwx(vfs_t *vfsp, int vtype, int mode, cred_t *cr)
1217 {
1218         /* See the secpolicy call below. */
1219         static const vnode_t tmpl_vdir = { .v_type = VDIR };
1220         static const vnode_t tmpl_vreg = { .v_type = VREG };
1221         vattr_t         va;
1222         vnode_t         *tvp;
1223         struct smbmntinfo *smi = VFTOSMI(vfsp);
1224         int shift = 0;
1225
1226         /*
1227          * Build our (fabricated) vnode attributes.
1228          * XXX: Could make these templates in the
1229          * per-mount struct and use them here.
1230          */
1231         bzero(&va, sizeof (va));
1232         va.va_mask = AT_TYPE | AT_MODE | AT_UID | AT_GID;
1233         va.va_type = vtype;
1234         va.va_mode = (vtype == VDIR) ?
1235             smi->smi_dmode : smi->smi_fmode;
1236         va.va_uid = smi->smi_uid;
1237         va.va_gid = smi->smi_gid;
1238
1239         /*
1240          * Disallow write attempts on read-only file systems,
1241          * unless the file is a device or fifo node.  Note:
1242          * Inline vn_is_readonly and IS_DEVVP here because
1243          * we may not have a vnode ptr.  Original expr. was:
1244          * (mode & VWRITE) && vn_is_readonly(vp) && !IS_DEVVP(vp))
1245          */
1246         if ((mode & VWRITE) &&
1247             (vfsp->vfs_flag & VFS_RDONLY) &&
1248             !(vtype == VCHR || vtype == VBLK || vtype == VFIFO))
1249                 return (EROFS);
1250
1251         /*
1252          * Disallow attempts to access mandatory lock files.
1253          * Similarly, expand MANDLOCK here.
1254          * XXX: not sure we need this.
1255          */
1256         if ((mode & (VWRITE | VREAD | VEXEC)) &&
1257             va.va_type == VREG && MANDMODE(va.va_mode))
```

```
1258                 return (EACCES);

1260         /*
1261          * Access check is based on only
1262          * one of owner, group, public.
1263          * If not owner, then check group.
1264          * If not a member of the group,
1265          * then check public access.
1266          */
1267         if (crgetuid(cr) != va.va_uid) {
1268                 shift += 3;
1269                 if (!groupmember(va.va_gid, cr))
1270                         shift += 3;
1271         }

1273         /*
1274          * We need a vnode for secpolicy_vnode_access,
1275          * but the only thing it looks at is v_type,
1276          * so pass one of the templates above.
1277          */
1278         tvp = (va.va_type == VDIR) ?
1279             (vnode_t *)&tmpl_vdir :
1280             (vnode_t *)&tmpl_vreg;

1282         return (secpolicy_vnode_access2(cr, tvp, va.va_uid,
1283             va.va_mode << shift, mode));
1284 }
_____unchanged_portion_omitted_

3001 /* ARGSUSED */
3002 static int
3003 smbfs_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
3004         caller_context_t *ct)
3005 {
3006         vfs_t *vfs;
3007         smbmntinfo_t *smi;
3008         struct smb_share *ssp;

3010         vfs = vp->v_vfsp;
3011         smi = VFTOSMI(vfs);

3013         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3014                 return (EIO);

3016         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3017                 return (EIO);

3019         switch (cmd) {
3020         case _PC_FILESIZEBITS:
3021                 ssp = smi->smi_share;
3022                 if (SSTOVC(ssp)->vc_sopt.sv_caps & SMB_CAP_LARGE_FILES)
3023                         *valp = 64;
3024                 else
3025                         *valp = 32;
3026                 break;

3028         case _PC_LINK_MAX:
3029                 /* We only ever report one link to an object */
3030                 *valp = 1;
3031                 break;

3033         case _PC_ACL_ENABLED:
3034                 /*
3035                  * Always indicate that ACLs are enabled and
3036                  * that we support ACE_T format, otherwise
3037                  * libsec will ask for ACLENT_T format data
```



```
3038                  * which we don't support.
3039                  */
3040                 *valp = _ACL_ACE_ENABLED;
3041                 break;

3043         case _PC_SYMLINK_MAX:   /* No symlinks until we do Unix extensions */
3044                 *valp = 0;
3045                 break;

3047         case _PC_XATTR_EXISTS:
3048                 if (vfs->vfs_flag & VFS_XATTR) {
3049                         *valp = smbfs_xa_exists(vp, cr);
3050                         break;
3051                 }
3052                 return (EINVAL);

3054         case _PC_SATTR_ENABLED:
3055         case _PC_SATTR_EXISTS:
3056                 *valp = 1;
3057                 break;

3059         case _PC_TIMESTAMP_RESOLUTION:
3060                 /*
3061                  * Windows times are tenths of microseconds
3062                  * (multiples of 100 nanoseconds).
3063                  */
3064                 *valp = 100L;
3065                 break;

3067         default:
3068                 return (fs_pathconf(vp, cmd, valp, cr, ct));
3069         }
3070         return (0);
3071 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   39409 Tue Aug 31 13:02:05 2010
new/usr/src/uts/common/fs/xattr.c
6975745 xattr directories should be more transparent
**********************************************************
______unchanged_portion_omitted_

  55 typedef struct {
  56         gfs_dir_t       xattr_gfs_private;
  57         vnode_t         *xattr_realvp;
  57         vnode_t         *xattr_realvp;  /* Only used for VOP_REALVP */
  58 } xattr_dir_t;

  60 /*
  61  * xattr_realvp is only used for VOP_REALVP, this is so we don't
  62  * keep an unnecessary hold on the *real* xattr dir unless we have
  63  * no other choice.
  64  */

  60 /* ARGSUSED */
  61 static int
  62 xattr_file_open(vnode_t **vpp, int flags, cred_t *cr, caller_context_t *ct)
  63 {
  64         xattr_file_t *np = (*vpp)->v_data;

  66         if ((np->xattr_view == XATTR_VIEW_READONLY) && (flags & FWRITE))
  67                 return (EACCES);

  69         return (0);
  70 }
______unchanged_portion_omitted_

 872 static int
 873 xattr_dir_realdir(vnode_t *dvp, vnode_t **realdvp, int lookup_flags,
 874     cred_t *cr, caller_context_t *ct)
 875 {
 876         xattr_dir_t *xattr_dir;
 882         vnode_t *pvp;
 877         int error;
 884         struct pathname pn;
 885         char *startnm = "";

 879         *realdvp = NULL;

 881         if (dvp->v_type != VDIR)
 882                 return (EINVAL);
 889         pvp = gfs_file_parent(dvp);

 884         mutex_enter(&dvp->v_lock);
 885         xattr_dir = dvp->v_data;
 886         *realdvp = xattr_dir->xattr_realvp;
 887         mutex_exit(&dvp->v_lock);
 891         error = pn_get(startnm, UIO_SYSSPACE, &pn);
 892         if (error) {
 893                 VN_RELE(pvp);
 894                 return (error);
 895         }

 889         if (*realdvp != NULL) {
 890                 VN_HOLD(*realdvp);
 891                 error = 0;
 892         } else
 893                 error = ENOENT;
 897         /*
 898          * Set the LOOKUP_HAVE_SYSATTR_DIR flag so that we don't get into an
 899          * infinite loop with fop_lookup calling back to xattr_dir_lookup.
```



```
 900          */
 901         lookup_flags |= LOOKUP_HAVE_SYSATTR_DIR;
 902         error = VOP_LOOKUP(pvp, startnm, realdvp, &pn, lookup_flags,
 903             rootvp, cr, ct, NULL, NULL);
 904         pn_free(&pn);

 895         return (error);
 896 }

 898 /* ARGSUSED */
 899 static int
 900 xattr_dir_open(vnode_t **vpp, int flags, cred_t *cr, caller_context_t *ct)
 901 {
 902         vnode_t *realvp;
 903         int error;

 905         if (flags & FWRITE) {
 906                 return (EACCES);
 907         }

 909         /*
 910          * The underlying FS may need this VOP call.
 911          */
 912         error = xattr_dir_realdir(*vpp, &realvp, LOOKUP_XATTR, cr, ct);
 913         if (error == 0) {
 914                 error = VOP_OPEN(&realvp, flags, cr, ct);
 915                 VN_RELE(realvp);
 916                 if (error)
 917                         return (error);
 918         } /* else ignore this error */

 920         return (0);
 921 }

 923 /* ARGSUSED */
 924 static int
 925 xattr_dir_close(vnode_t *vp, int flags, int count, offset_t off, cred_t *cr,
 922 xattr_dir_close(vnode_t *vpp, int flags, int count, offset_t off, cred_t *cr,
 926     caller_context_t *ct)
 927 {
 928         vnode_t *realvp;
 929         int error;

 931         /*
 932          * The underlying FS may need this VOP call.
 933          */
 934         error = xattr_dir_realdir(vp, &realvp, LOOKUP_XATTR, cr, ct);
 935         if (error == 0) {
 936                 error = VOP_CLOSE(realvp, flags, count, off, cr, ct);
 937                 VN_RELE(realvp);
 938                 if (error)
 939                         return (error);
 940         } /* else ignore this error */

 942         return (0);
 943 }
______unchanged_portion_omitted_

1372 /* ARGSUSED */
1373 static int
1374 xattr_dir_realvp(vnode_t *vp, vnode_t **realvp, caller_context_t *ct)
1375 {
1359         xattr_dir_t *xattr_dir;

1361         mutex_enter(&vp->v_lock);
1362         xattr_dir = vp->v_data;
```

```
1363         if (xattr_dir->xattr_realvp) {
1364                 *realvp = xattr_dir->xattr_realvp;
1365                 mutex_exit(&vp->v_lock);
1366                 return (0);
1367         } else {
1368                 vnode_t *xdvp;
1376         int error;

1378         error = xattr_dir_realdir(vp, realvp, LOOKUP_XATTR, kcred, NULL);
1371                 mutex_exit(&vp->v_lock);
1372                 if ((error = xattr_dir_realdir(vp, &xdvp,
1373                     LOOKUP_XATTR, kcred, NULL)) == 0) {
1374                         /*
1375                          * verify we aren't racing with another thread
1376                          * to find the xattr_realvp
1377                          */
1378                         mutex_enter(&vp->v_lock);
1379                         if (xattr_dir->xattr_realvp == NULL) {
1380                                 xattr_dir->xattr_realvp = xdvp;
1381                                 *realvp = xdvp;
1382                                 mutex_exit(&vp->v_lock);
1383                         } else {
1384                                 *realvp = xattr_dir->xattr_realvp;
1385                                 mutex_exit(&vp->v_lock);
1386                                 VN_RELE(xdvp);
1387                         }
1388                 }
1379         return (error);

1390         }
1381 }
______unchanged_portion_omitted_

1463 /* See vnode.c: fop_lookup() */
1464 int
1465 xattr_dir_lookup(vnode_t *dvp, vnode_t **vpp, int flags, cred_t *cr)
1466 {
1467         int error = 0;
1468         vnode_t *gfs_vp = NULL;
1469         vnode_t *real_vp = NULL;
1470         xattr_dir_t *xattr_dir;
1471         struct pathname pn;
1472         char *nm = "";

1474         *vpp = NULL;

1476         if (dvp->v_type != VDIR && dvp->v_type != VREG)
1477                 return (EINVAL);

1479         mutex_enter(&dvp->v_lock);

1481         /*
1482          * If we're already in sysattr space, don't allow creation
1483          * of another level of sysattrs.
1484          */
1485         if (dvp->v_flag & V_SYSATTR) {
1486                 mutex_exit(&dvp->v_lock);
1487                 return (EINVAL);
1488         }

1490         if (dvp->v_xattrdir != NULL) {
1491                 gfs_vp = dvp->v_xattrdir;
1492                 VN_HOLD(gfs_vp);
1495                 *vpp = dvp->v_xattrdir;
1496                 VN_HOLD(*vpp);
1493         } else {
```



```
1494                 ulong_t val;
1495                 int xattrs_allowed = dvp->v_vfsp->vfs_flag & VFS_XATTR;
1496                 int sysattrs_allowed = 1;

1498                 /*
1499                  * We have to drop the lock on dvp.  gfs_dir_create will
1500                  * grab it for a VN_HOLD.
1501                  */
1502                 mutex_exit(&dvp->v_lock);

1504                 /*
1505                  * If dvp allows xattr creation, but not sysattr
1506                  * creation, return the real xattr dir vp. We can't
1507                  * use the vfs feature mask here because _PC_SATTR_ENABLED
1508                  * has vnode-level granularity (e.g. .zfs).
1509                  */
1510                 error = VOP_PATHCONF(dvp, _PC_SATTR_ENABLED, &val, cr, NULL);
1511                 if (error != 0 || val == 0)
1512                         sysattrs_allowed = 0;

1514                 if (!xattrs_allowed && !sysattrs_allowed)
1515                         return (EINVAL);

1517                 if (!sysattrs_allowed) {
1522                         struct pathname pn;
1523                         char *nm = "";

1518                         error = pn_get(nm, UIO_SYSSPACE, &pn);
1519                         if (error)
1520                                 return (error);
1521                         error = VOP_LOOKUP(dvp, nm, vpp, &pn,
1522                             flags|LOOKUP_HAVE_SYSATTR_DIR, rootvp, cr, NULL,
1523                             NULL, NULL);
1524                         pn_free(&pn);
1525                         return (error);
1526                 }

1528                 /*
1529                  * Note that we act as if we were given CREATE_XATTR_DIR,
1530                  * but only for creation of the GFS directory.
1531                  */
1532                 gfs_vp = gfs_dir_create(
1539                 *vpp = gfs_dir_create(
1533                     sizeof (xattr_dir_t), dvp, xattr_dir_ops, xattr_dirents,
1534                     xattrdir_do_ino, MAXNAMELEN, NULL, xattr_lookup_cb);
1535                 mutex_enter(&dvp->v_lock);
1536                 if (dvp->v_xattrdir != NULL) {
1537                         /*
1538                          * We lost the race to create the xattr dir.
1539                          * Destroy this one, use the winner.  We can't
1540                          * just call VN_RELE(*vpp), because the vnode
1541                          * is only partially initialized.
1542                          */
1543                         gfs_dir_t *dp = gfs_vp->v_data;
1550                         gfs_dir_t *dp = (*vpp)->v_data;

1545                         ASSERT(gfs_vp->v_count == 1);
1546                         vn_free(gfs_vp);
1552                         ASSERT((*vpp)->v_count == 1);
1553                         vn_free(*vpp);

1548                         mutex_destroy(&dp->gfsd_lock);
1549                         kmem_free(dp->gfsd_static,
1550                             dp->gfsd_nstatic * sizeof (gfs_dirent_t));
1551                         kmem_free(dp, dp->gfsd_file.gfs_size);
```

```
                        /*
                         * There is an implied VN_HOLD(dvp) here.  We should
                         * be doing a VN_RELE(dvp) to clean up the reference
                         * from gfs_vp, and then a VN_HOLD(dvp) for the new
                         * from *vpp, and then a VN_HOLD(dvp) for the new
                         * reference.  Instead, we just leave the count alone.
                         */

                        gfs_vp = dvp->v_xattrdir;
                        VN_HOLD(gfs_vp);
                        *vpp = dvp->v_xattrdir;
                        VN_HOLD(*vpp);
                } else {
                        gfs_vp->v_flag |= (V_XATTRDIR|V_SYSATTR);
                        dvp->v_xattrdir = gfs_vp;
                        (*vpp)->v_flag |= (V_XATTRDIR|V_SYSATTR);
                        dvp->v_xattrdir = *vpp;
                }
        }
        mutex_exit(&dvp->v_lock);

        /*
         * In order to make this module relatively transparent
         * to the underlying filesystem, we need to lookup the
         * xattr dir in the lower filesystem and (if found)
         * keep a hold on it for as long as there is a hold
         * on the gfs_vp we're about to return.  This hold is
         * released in xattr_dir_inactive.
         */
        xattr_dir = gfs_vp->v_data;
        if ((dvp->v_vfsp->vfs_flag & VFS_XATTR) &&
            (xattr_dir->xattr_realvp == NULL)) {
                error = pn_get(nm, UIO_SYSSPACE, &pn);
                if (error == 0) {
                        error = VOP_LOOKUP(dvp, nm, &real_vp, &pn,
                            flags|LOOKUP_HAVE_SYSATTR_DIR, rootvp, cr, NULL,
                            NULL, NULL);
                        pn_free(&pn);
                }
                if (error == 0) {
                        mutex_enter(&gfs_vp->v_lock);
                        if (xattr_dir->xattr_realvp == NULL)
                                xattr_dir->xattr_realvp = real_vp;
                        else
                                VN_RELE(real_vp);
                        mutex_exit(&gfs_vp->v_lock);
                }
        }

        *vpp = gfs_vp;
        return (0);
        return (error);
1599 }
```

_____unchanged_portion_omitted_