```
**********************************************************
   16706 Sat Aug 18 10:48:41 2012
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_client.c
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

 120 /*
 121  * Purge all of the various data caches.
 122  */
 123 /*ARGSUSED*/
 124 void
 125 smbfs_purge_caches(struct vnode *vp)
 126 {
 127 #if 0   /* not yet: mmap support */
 127         /*
 128          * NFS: Purge the DNLC for this vp,
 129          * Clear any readdir state bits,
 130          * the readlink response cache, ...
 131          */
 132         smbnode_t *np = VTOSMB(vp);

 134         /*
 135          * Flush the page cache.
 136          */
 137         if (vn_has_cached_data(vp)) {
 138                 (void) VOP_PUTPAGE(vp, (u_offset_t) 0, 0, B_INVAL, np->r_cred, N
 139                 (void) VOP_PUTPAGE(vp, (u_offset_t)0, 0, B_INVAL, cr, NULL);
 139         }
 141 #endif  /* not yet */
 140 }
______unchanged_portion_omitted_
 199 #endif  /* not yet */

 201 /*
 202  * Set attributes cache for given vnode using SMB fattr
 203  * and update the attribute cache timeout.
 204  *
 205  * From NFS: nfs_attrcache, nfs_attrcache_va
 206  */
 207 void
 208 smbfs_attrcache_fa(vnode_t *vp, struct smbfattr *fap)
 209 {
 210         smbnode_t *np;
 211         smbmntinfo_t *smi;
 212         hrtime_t delta, now;
 213         u_offset_t newsize;
 214         vtype_t  vtype, oldvt;
 215         mode_t mode;

 217         np = VTOSMB(vp);
 218         smi = VTOSMI(vp);

 220         /*
 221          * We allow v_type to change, so set that here
 222          * (and the mode, which depends on the type).
 223          */
 224         if (fap->fa_attr & SMB_FA_DIR) {
 225                 vtype = VDIR;
 226                 mode = smi->smi_dmode;
 227         } else {
 228                 vtype = VREG;
 229                 mode = smi->smi_fmode;
 230         }

 232         mutex_enter(&np->r_statelock);
```



```
 233         now = gethrtime();

 235         /*
 236          * Delta is the number of nanoseconds that we will
 237          * cache the attributes of the file.  It is based on
 238          * the number of nanoseconds since the last time that
 239          * we detected a change.  The assumption is that files
 240          * that changed recently are likely to change again.
 241          * There is a minimum and a maximum for regular files
 242          * and for directories which is enforced though.
 243          *
 244          * Using the time since last change was detected
 245          * eliminates direct comparison or calculation
 246          * using mixed client and server times.  SMBFS
 247          * does not make any assumptions regarding the
 248          * client and server clocks being synchronized.
 249          */
 250         if (fap->fa_mtime.tv_sec  != np->r_attr.fa_mtime.tv_sec ||
 251             fap->fa_mtime.tv_nsec != np->r_attr.fa_mtime.tv_nsec ||
 252             fap->fa_size          != np->r_attr.fa_size)
 253                 np->r_mtime = now;

 255         if ((smi->smi_flags & SMI_NOAC) || (vp->v_flag & VNOCACHE))
 256                 delta = 0;
 257         else {
 258                 delta = now - np->r_mtime;
 259                 if (vtype == VDIR) {
 260                         if (delta < smi->smi_acdirmin)
 261                                 delta = smi->smi_acdirmin;
 262                         else if (delta > smi->smi_acdirmax)
 263                                 delta = smi->smi_acdirmax;
 264                 } else {
 265                         if (delta < smi->smi_acregmin)
 266                                 delta = smi->smi_acregmin;
 267                         else if (delta > smi->smi_acregmax)
 268                                 delta = smi->smi_acregmax;
 269                 }
 270         }

 272         np->r_attrtime = now + delta;
 273         np->r_attr = *fap;
 274         np->n_mode = mode;
 275         oldvt = vp->v_type;
 276         vp->v_type = vtype;

 278         /*
 279          * Shall we update r_size? (local notion of size)
 280          *
 281          * The real criteria for updating r_size should be:
 282          * if the file has grown on the server, or if
 283          * the client has not modified the file.
 284          *
 285          * Also deal with the fact that SMB presents
 286          * directories as having size=0.  Doing that
 287          * here and leaving fa_size as returned OtW
 288          * avoids fixing the size lots of places.
 289          */
 290         newsize = fap->fa_size;
 291         if (vtype == VDIR && newsize < DEV_BSIZE)
 292                 newsize = DEV_BSIZE;

 294         if (np->r_size != newsize) {
 295                 if (!vn_has_cached_data(vp)
 296                     || (!(np->r_flags & RDIRTY)&& np->r_count == 0)) {
 297 #if 0   /* not yet: mmap support */
 298                 if (!vn_has_cached_data(vp) || ...)
```

```
299                         /* XXX: See NFS page cache code. */
300 #endif  /* not yet */
297                         /* OK to set the size. */
298                         np->r_size = newsize;
299                 }
300         }
301 #endif /* ! codereview */

303         /* NFS: np->r_flags &= ~RWRITEATTR; */
304         np->n_flag &= ~NATTRCHANGED;

306         mutex_exit(&np->r_statelock);

308         if (oldvt != vtype) {
309                 SMBVDEBUG("vtype change %d to %d\n", oldvt, vtype);
310         }
311 }

313 /*
314  * Fill in attribute from the cache.
315  *
316  * If valid, copy to *fap and return zero,
317  * otherwise return an error.
318  *
319  * From NFS: nfs_getattr_cache()
320  */
321 int
322 smbfs_getattr_cache(vnode_t *vp, struct smbfattr *fap)
323 {
324         smbnode_t *np;
325         int error;

327         np = VTOSMB(vp);

329         mutex_enter(&np->r_statelock);
330         if (gethrtime() >= np->r_attrtime) {
331                 /* cache expired */
332                 error = ENOENT;
333         } else {
334                 /* cache is valid */
335                 *fap = np->r_attr;
336                 error = 0;
337         }
338         mutex_exit(&np->r_statelock);

340         return (error);
341 }

343 /*
344  * Get attributes over-the-wire and update attributes cache
345  * if no error occurred in the over-the-wire operation.
346  * Return 0 if successful, otherwise error.
347  * From NFS: nfs_getattr_otw
348  */
349 int
350 smbfs_getattr_otw(vnode_t *vp, struct smbfattr *fap, cred_t *cr)
351 {
352         struct smbnode *np;
353         struct smb_cred scred;
354         int error;

356         np = VTOSMB(vp);

358         /*
359          * NFS uses the ACL rpc here (if smi_flags & SMI_ACL)
360          * With SMB, getting the ACL is a significantly more
```



```
361          * expensive operation, so we do that only when asked
362          * for the uid/gid.  See smbfsgetattr().
363          */

365         /* Shared lock for (possible) n_fid use. */
366         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
367                 return (EINTR);
368         smb_credinit(&scred, cr);

370         bzero(fap, sizeof (*fap));
371         error = smbfs_smb_getfattr(np, fap, &scred);

373         smb_credrele(&scred);
374         smbfs_rw_exit(&np->r_lkserlock);

376         if (error) {
377                 /* NFS had: PURGE_STALE_FH(error, vp, cr) */
378                 smbfs_attrcache_remove(np);
379                 if (error == ENOENT || error == ENOTDIR) {
380                         /*
381                          * Getattr failed because the object was
382                          * removed or renamed by another client.
383                          * Remove any cached attributes under it.
384                          */
385                         smbfs_attrcache_prune(np);
386                 }
387                 return (error);
388         }

390         /*
391          * NFS: smbfs_cache_fattr(vap, fa, vap, t, cr);
392          * which did: fattr_to_vattr, nfs_attr_cache.
393          * We cache the fattr form, so just do the
394          * cache check and store the attributes.
395          */
396         smbfs_cache_check(vp, fap);
397         smbfs_attrcache_fa(vp, fap);

399         return (0);
400 }

402 /*
403  * Return either cached or remote attributes. If get remote attr
404  * use them to check and invalidate caches, then cache the new attributes.
405  *
406  * From NFS: nfsgetattr()
407  */
408 int
409 smbfsgetattr(vnode_t *vp, struct vattr *vap, cred_t *cr)
410 {
411         struct smbfattr fa;
412         smbmntinfo_t *smi;
413         uint_t mask;
414         int error;

416         smi = VTOSMI(vp);

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

420         /*
421          * If asked for UID or GID, update n_uid, n_gid.
422          */
423         mask = AT_ALL;
424         if (vap->va_mask & (AT_UID | AT_GID)) {
425                 if (smi->smi_flags & SMI_ACL)
426                         (void) smbfs_acl_getids(vp, cr);
```

```
427                 /* else leave as set in make_smbnode */
428         } else {
429                 mask &= ~(AT_UID | AT_GID);
430         }
431
432         /*
433          * If we've got cached attributes, just use them;
434          * otherwise go to the server to get attributes,
435          * which will update the cache in the process.
436          */
437         error = smbfs_getattr_cache(vp, &fa);
438         if (error)
439                 error = smbfs_getattr_otw(vp, &fa, cr);
440         if (error)
441                 return (error);
442
443         /*
444          * Re. client's view of the file size, see:
445          * smbfs_attrcache_fa, smbfs_getattr_otw
446          */
447
448         error = smbfattr_to_vattr(vp, &fa, vap);
449         vap->va_mask = mask;
450
451         return (error);
452 }
453
454
455 /*
456  * Convert SMB over the wire attributes to vnode form.
457  * Returns 0 for success, error if failed (overflow, etc).
458  * From NFS: nattr_to_vattr()
459  */
460 int
461 smbfattr_to_vattr(vnode_t *vp, struct smbfattr *fa, struct vattr *vap)
462 {
463         struct smbnode *np = VTOSMB(vp);
464
465         /* Set va_mask in caller */
466
467         /*
468          * Take type, mode, uid, gid from the smbfs node,
469          * which has have been updated by _getattr_otw.
470          */
471         vap->va_type = vp->v_type;
472         vap->va_mode = np->n_mode;
473
474         vap->va_uid = np->n_uid;
475         vap->va_gid = np->n_gid;
476
477         vap->va_fsid = vp->v_vfsp->vfs_dev;
478         vap->va_nodeid = np->n_ino;
479         vap->va_nlink = 1;
480
481         /*
482          * Difference from NFS here:  We cache attributes as
483          * reported by the server, so r_attr.fa_size is the
484          * server's idea of the file size.  This is called
485          * for getattr, so we want to return the client's
486          * idea of the file size.  NFS deals with that in
487          * nfsgetattr(), the equivalent of our caller.
488          */
489         vap->va_size = np->r_size;
490
491         /*
492          * Times.  Note, already converted from NT to
```



```
493          * Unix form (in the unmarshalling code).
494          */
495         vap->va_atime = fa->fa_atime;
496         vap->va_mtime = fa->fa_mtime;
497         vap->va_ctime = fa->fa_ctime;
498
499         /*
500          * rdev, blksize, seq are made up.
501          * va_nblocks is 512 byte blocks.
502          */
503         vap->va_rdev = vp->v_rdev;
504         vap->va_blksize = MAXBSIZE;
505         vap->va_nblocks = (fsblkcnt64_t)btod(np->r_attr.fa_allocsz);
506         vap->va_seq = 0;
507
508         return (0);
509 }
510
511
512 /*
513  * SMB Client initialization and cleanup.
514  * Much of it is per-zone now.
515  */
516
517
518 /* ARGSUSED */
519 static void *
520 smbfs_zone_init(zoneid_t zoneid)
521 {
522         smi_globals_t *smg;
523
524         smg = kmem_alloc(sizeof (*smg), KM_SLEEP);
525         mutex_init(&smg->smg_lock, NULL, MUTEX_DEFAULT, NULL);
526         list_create(&smg->smg_list, sizeof (smbmntinfo_t),
527             offsetof(smbmntinfo_t, smi_zone_node));
528         smg->smg_destructor_called = B_FALSE;
529         return (smg);
530 }
531
532 /*
533  * Callback routine to tell all SMBFS mounts in the zone to stop creating new
534  * threads.  Existing threads should exit.
535  */
536 /* ARGSUSED */
537 static void
538 smbfs_zone_shutdown(zoneid_t zoneid, void *data)
539 {
540         smi_globals_t *smg = data;
541         smbmntinfo_t *smi;
542
543         ASSERT(smg != NULL);
544 again:
545         mutex_enter(&smg->smg_lock);
546         for (smi = list_head(&smg->smg_list); smi != NULL;
547             smi = list_next(&smg->smg_list, smi)) {
548
549                 /*
550                  * If we've done the shutdown work for this FS, skip.
551                  * Once we go off the end of the list, we're done.
552                  */
553                 if (smi->smi_flags & SMI_DEAD)
554                         continue;
555
556                 /*
557                  * We will do work, so not done.  Get a hold on the FS.
558                  */
```

```
559                 VFS_HOLD(smi->smi_vfsp);
560
561                 mutex_enter(&smi->smi_lock);
562                 smi->smi_flags |= SMI_DEAD;
563                 mutex_exit(&smi->smi_lock);
564
565                 /*
566                  * Drop lock and release FS, which may change list, then repeat.
567                  * We're done when every mi has been done or the list is empty.
568                  */
569                 mutex_exit(&smg->smg_lock);
570                 VFS_RELE(smi->smi_vfsp);
571                 goto again;
572         }
573         mutex_exit(&smg->smg_lock);
574 }
575
576 static void
577 smbfs_zone_free_globals(smi_globals_t *smg)
578 {
579         list_destroy(&smg->smg_list);   /* makes sure the list is empty */
580         mutex_destroy(&smg->smg_lock);
581         kmem_free(smg, sizeof (*smg));
582
583 }
584
585 /* ARGSUSED */
586 static void
587 smbfs_zone_destroy(zoneid_t zoneid, void *data)
588 {
589         smi_globals_t *smg = data;
590
591         ASSERT(smg != NULL);
592         mutex_enter(&smg->smg_lock);
593         if (list_head(&smg->smg_list) != NULL) {
594                 /* Still waiting for VFS_FREEVFS() */
595                 smg->smg_destructor_called = B_TRUE;
596                 mutex_exit(&smg->smg_lock);
597                 return;
598         }
599         smbfs_zone_free_globals(smg);
600 }
601
602 /*
603  * Add an SMBFS mount to the per-zone list of SMBFS mounts.
604  */
605 void
606 smbfs_zonelist_add(smbmntinfo_t *smi)
607 {
608         smi_globals_t *smg;
609
610         smg = zone_getspecific(smi_list_key, smi->smi_zone_ref.zref_zone);
611         mutex_enter(&smg->smg_lock);
612         list_insert_head(&smg->smg_list, smi);
613         mutex_exit(&smg->smg_lock);
614 }
615
616 /*
617  * Remove an SMBFS mount from the per-zone list of SMBFS mounts.
618  */
619 void
620 smbfs_zonelist_remove(smbmntinfo_t *smi)
621 {
622         smi_globals_t *smg;
623
624         smg = zone_getspecific(smi_list_key, smi->smi_zone_ref.zref_zone);
```



```
625         mutex_enter(&smg->smg_lock);
626         list_remove(&smg->smg_list, smi);
627         /*
628          * We can be called asynchronously by VFS_FREEVFS() after the zone
629          * shutdown/destroy callbacks have executed; if so, clean up the zone's
630          * smi_globals.
631          */
632         if (list_head(&smg->smg_list) == NULL &&
633             smg->smg_destructor_called == B_TRUE) {
634                 smbfs_zone_free_globals(smg);
635                 return;
636         }
637         mutex_exit(&smg->smg_lock);
638 }
639
640 #ifdef  lint
641 #define NEED_SMBFS_CALLBACKS    1
642 #endif
643
644 #ifdef NEED_SMBFS_CALLBACKS
645 /*
646  * Call-back hooks for netsmb, in case we want them.
647  * Apple's VFS wants them.  We may not need them.
648  */
649 /*ARGSUSED*/
650 static void smbfs_dead(smb_share_t *ssp)
651 {
652         /*
653          * Walk the mount list, finding all mounts
654          * using this share...
655          */
656 }
657
658 /*ARGSUSED*/
659 static void smbfs_cb_nop(smb_share_t *ss)
660 {
661         /* no-op */
662 }
663
664 smb_fscb_t smbfs_cb = {
665         .fscb_disconn   = smbfs_dead,
666         .fscb_connect   = smbfs_cb_nop,
667         .fscb_down      = smbfs_cb_nop,
668         .fscb_up        = smbfs_cb_nop };
669
670 #endif /* NEED_SMBFS_CALLBACKS */
671
672 /*
673  * SMBFS Client initialization routine.  This routine should only be called
674  * once.  It performs the following tasks:
675  *      - Initalize all global locks
676  *      - Call sub-initialization routines (localize access to variables)
677  */
678 int
679 smbfs_clntinit(void)
680 {
681
682         zone_key_create(&smi_list_key, smbfs_zone_init, smbfs_zone_shutdown,
683             smbfs_zone_destroy);
684 #ifdef NEED_SMBFS_CALLBACKS
685         (void) smb_fscb_set(&smbfs_cb);
686 #endif /* NEED_SMBFS_CALLBACKS */
687         return (0);
688 }
689
690 /*
```

```
691  * This routine is called when the modunload is called. This will cleanup
692  * the previously allocated/initialized nodes.
693  */
694 void
695 smbfs_clntfini(void)
696 {
697 #ifdef NEED_SMBFS_CALLBACKS
698         (void) smb_fscb_set(NULL);
699 #endif /* NEED_SMBFS_CALLBACKS */
700         (void) zone_key_delete(smi_list_key);
701 }
```

```
**********************************************************
   11749 Sat Aug 18 10:48:42 2012
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_node.h
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_

 125 /*
 126  * Below is the SMBFS-specific representation of a "node".
 127  * This struct is a mixture of Sun NFS and Darwin code.
 128  * Fields starting with "r_" came from NFS struct "rnode"
 129  * and fields starting with "n_" came from Darwin, or
 130  * were added during the Solaris port.  We have avoided
 131  * renaming fields so we would not cause excessive
 132  * changes in the code using this struct.
 133  *
 134  * Now using an AVL tree instead of hash lists, but kept the
 135  * "hash" in some member names and functions to reduce churn.
 136  * One AVL tree per mount replaces the global hash buckets.
 137  *
 138  * Notes carried over from the NFS code:
 139  *
 140  * The smbnode is the "inode" for remote files.  It contains all the
 141  * information necessary to handle remote file on the client side.
 142  *
 143  * Note on file sizes:  we keep two file sizes in the smbnode: the size
 144  * according to the client (r_size) and the size according to the server
 145  * (r_attr.fa_size).  They can differ because we modify r_size during a
 146  * write system call (smbfs_rdwr), before the write request goes over the
 147  * wire (before the file is actually modified on the server).  If an OTW
 148  * request occurs before the cached data is written to the server the file
 149  * size returned from the server (r_attr.fa_size) may not match r_size.
 150  * r_size is the one we use, in general.  r_attr.fa_size is only used to
 151  * determine whether or not our cached data is valid.
 152  *
 153  * Each smbnode has 3 locks associated with it (not including the smbnode
 154  * "hash" AVL tree and free list locks):
 155  *
 156  *      r_rwlock:       Serializes smbfs_write and smbfs_setattr requests
 157  *                      and allows smbfs_read requests to proceed in parallel.
 158  *                      Serializes reads/updates to directories.
 159  *
 160  *      r_lkserlock:    Serializes lock requests with map, write, and
 161  *                      readahead operations.
 162  *
 163  *      r_statelock:    Protects all fields in the smbnode except for
 164  *                      those listed below.  This lock is intented
 165  *                      to be held for relatively short periods of
 166  *                      time (not accross entire putpage operations,
 167  *                      for example).
 168  *
 169  * The following members are protected by the mutex smbfreelist_lock:
 170  *      r_freef
 171  *      r_freeb
 172  *
 173  * The following members are protected by the AVL tree rwlock:
 174  *      r_avl_node      (r__hdr.hdr_avl_node)
 175  *
 176  * Note: r_modaddr is only accessed when the r_statelock mutex is held.
 177  *      Its value is also controlled via r_rwlock.  It is assumed that
 178  *      there will be only 1 writer active at a time, so it safe to
 179  *      set r_modaddr and release r_statelock as long as the r_rwlock
 180  *      writer lock is held.
 181  *
 182  * 64-bit offsets: the code formerly assumed that atomic reads of
 183  * r_size were safe and reliable; on 32-bit architectures, this is
```



```
 184  * not true since an intervening bus cycle from another processor
 185  * could update half of the size field.  The r_statelock must now
 186  * be held whenever any kind of access of r_size is made.
 187  *
 188  * Lock ordering:
 189  *      r_rwlock > r_lkserlock > r_statelock
 190  */

 192 typedef struct smbnode {
 193         /* Our linkage in the node cache AVL tree (see above). */
 194         smbfs_node_hdr_t        r__hdr;

 196         /* short-hand names for r__hdr members */
 197 #define r_avl_node      r__hdr.hdr_avl_node
 198 #define n_rpath         r__hdr.hdr_n_rpath
 199 #define n_rplen         r__hdr.hdr_n_rplen

 201         smbmntinfo_t    *n_mount;       /* VFS data */
 202         vnode_t         *r_vnode;       /* associated vnode */

 204         /*
 205          * Linkage in smbfreelist, for reclaiming nodes.
 206          * Lock for the free list is: smbfreelist_lock
 207          */
 208         struct smbnode  *r_freef;       /* free list forward pointer */
 209         struct smbnode  *r_freeb;       /* free list back pointer */

 211         smbfs_rwlock_t  r_rwlock;       /* serialize write/setattr requests */
 212         smbfs_rwlock_t  r_lkserlock;    /* serialize lock with other ops */
 213         kmutex_t        r_statelock;    /* protect (most) smbnode fields */

 215         /*
 216          * File handle, directory search handle,
 217          * and reference counts for them, etc.
 218          * Lock for these is: r_lkserlock
 219          */
 220         int             n_dirrefs;
 221         struct smbfs_fctx       *n_dirseq;      /* ff context */
 222         int             n_dirofs;       /* last ff offset */
 223         int             n_fidrefs;
 224         uint16_t        n_fid;          /* file handle */
 225         enum vtype      n_ovtype;       /* vnode type opened */
 226         uint32_t        n_rights;       /* granted rights */
 227         int             n_vcgenid;      /* gereration no. (reconnect) */

 229         /*
 230          * Misc. bookkeeping
 231          */
 232         cred_t          *r_cred;        /* current credentials */
 233         u_offset_t      r_nextr;        /* next read offset (read-ahead) */
 234         long            r_mapcnt;       /* count of mmapped pages */
 235         uint_t          r_count;        /* # of refs not reflect in v_count */
 236         uint_t          r_awcount;      /* # of outstanding async write */
 237         uint_t          r_gcount;       /* getattrs waiting to flush pages */
 238         u_offset_t      r_modaddr;      /* address for page in writenp */
 239 #endif /* ! codereview */
 240         uint_t          r_flags;        /* flags, see below */
 241         uint32_t        n_flag;         /* NXXX flags below */
 242         uint_t          r_error;        /* async write error */
 243         kcondvar_t      r_cv;           /* condvar for blocked threads */
 244         avl_tree_t      r_dir;          /* cache of readdir responses */
 245         rddir_cache     *r_direof;      /* pointer to the EOF entry */
 246         kthread_t       *r_serial;      /* id of purging thread */
 247         list_t          r_indelmap;     /* list of delmap callers */

 249         /*
```

```
250          * Attributes: local, and as last seen on the server.
251          * See notes above re: r_size vs r_attr.fa_size, etc.
252          */
253         smbfattr_t      r_attr;         /* attributes from the server */
254         hrtime_t        r_attrtime;     /* time attributes become invalid */
255         hrtime_t        r_mtime;        /* client time file last modified */
256         len_t           r_size;         /* client's view of file size */

258         /*
259          * Security attributes.
260          */
261         vsecattr_t      r_secattr;
262         hrtime_t        r_sectime;

264         /*
265          * Other attributes, not carried in smbfattr_t
266          */
267         u_longlong_t    n_ino;
268         uid_t           n_uid;
269         gid_t           n_gid;
270         mode_t          n_mode;
271 } smbnode_t;

273 /*
274  * Flag bits in: smbnode_t .n_flag
275  */
276 #define NFLUSHINPROG    0x00001
277 #define NFLUSHWANT      0x00002 /* they should gone ... */
278 #define NMODIFIED       0x00004 /* bogus, until async IO implemented */
279 #define NREFPARENT      0x00010 /* node holds parent from recycling */
280 #define NGOTIDS         0x00020
281 #define NRDIRSERIAL     0x00080 /* serialize readdir operation */
282 #define NISMAPPED       0x00800
283 #define NFLUSHWIRE      0x01000
284 #define NATTRCHANGED    0x02000 /* kill cached attributes at close */
285 #define NALLOC          0x04000 /* being created */
286 #define NWALLOC         0x08000 /* awaiting creation */
287 #define N_XATTR         0x10000 /* extended attribute (dir or file) */

289 /*
290  * Flag bits in: smbnode_t .r_flags
291  */
292 #define RREADDIRPLUS    0x1     /* issue a READDIRPLUS instead of READDIR */
293 #define RDIRTY          0x2     /* dirty pages from write operation */
294 #define RSTALE          0x4     /* file handle is stale */
295 #define RMODINPROGRESS  0x8     /* page modification happening */
296 #define RTRUNCATE       0x10    /* truncating, don't commit */
297 #define RHAVEVERF       0x20    /* have a write verifier to compare against */
298 #define RCOMMIT         0x40    /* commit in progress */
299 #define RCOMMITWAIT     0x80    /* someone is waiting to do a commit */
300 #define RHASHED         0x100   /* smbnode is in the "hash" AVL tree */
301 #define ROUTOFSPACE     0x200   /* an out of space error has happened */
302 #define RDIRECTIO       0x400   /* bypass the buffer cache */
303 #define RLOOKUP         0x800   /* a lookup has been performed */
304 #define RWRITEATTR      0x1000  /* attributes came from WRITE */
305 #define RINDNLCPURGE    0x2000  /* in the process of purging DNLC references */
306 #define RDELMAPLIST     0x4000  /* delmap callers tracking for as callback */

308 /*
309  * Convert between vnode and smbnode
310  */
311 #define VTOSMB(vp)      ((smbnode_t *)((vp)->v_data))
312 #define SMBTOV(np)      ((np)->r_vnode)

314 /*
315  * A macro to compute the separator that should be used for
```



```
316  * names under some directory.  See smbfs_fullpath().
317  */
318 #define SMBFS_DNP_SEP(dnp) \
319         (((dnp->n_flag & N_XATTR) == 0 && dnp->n_rplen > 1) ? '\\' : '\0')

321 #ifdef __cplusplus
322 }
323 #endif

325 #endif /* _FS_SMBFS_NODE_H_ */
```

```
**********************************************************
   10945 Sat Aug 18 10:48:43 2012
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr.h
*** NO COMMENTS ***
**********************************************************
______unchanged_portion_omitted_
 149 typedef struct smbfs_fctx smbfs_fctx_t;

 151 #define f_rq    f_urq.uf_rq
 152 #define f_t2    f_urq.uf_t2

 154 /*
 155  * smb level (smbfs_smb.c)
 156  */
 157 int  smbfs_smb_lock(struct smbnode *np, int op, caddr_t id,
 158         offset_t start, uint64_t len,   int largelock,
 159         struct smb_cred *scrp, uint32_t timeout);
 160 int  smbfs_smb_qfsattr(struct smb_share *ssp, struct smb_fs_attr_info *,
 161         struct smb_cred *scrp);
 162 int  smbfs_smb_statfs(struct smb_share *ssp, statvfs64_t *sbp,
 163         struct smb_cred *scrp);
 164 int  smbfs_smb_setfsize(struct smbnode *np, uint16_t fid, uint64_t newsize,
 165         struct smb_cred *scrp);

 167 int  smbfs_smb_getfattr(struct smbnode *np, struct smbfattr *fap,
 168         struct smb_cred *scrp);

 170 int  smbfs_smb_setfattr(struct smbnode *np, int fid,
 171         uint32_t attr, struct timespec *mtime, struct timespec *atime,
 172         struct smb_cred *scrp);

 174 int  smbfs_smb_open(struct smbnode *np, const char *name, int nmlen,
 175         int xattr, uint32_t rights, struct smb_cred *scrp,
 176         uint16_t *fidp, uint32_t *rightsp, struct smbfattr *fap);
 177 int  smbfs_smb_tmpopen(struct smbnode *np, uint32_t rights,
 178         struct smb_cred *scrp, uint16_t *fidp);
 179 int  smbfs_smb_close(struct smb_share *ssp, uint16_t fid,
 180         struct timespec *mtime, struct smb_cred *scrp);
 181 int  smbfs_smb_tmpclose(struct smbnode *ssp, uint16_t fid,
 182         struct smb_cred *scrp);
 183 int  smbfs_smb_create(struct smbnode *dnp, const char *name, int nmlen,
 184         int xattr, uint32_t disp, struct smb_cred *scrp, uint16_t *fidp);
 185 int  smbfs_smb_delete(struct smbnode *np, struct smb_cred *scrp,
 186         const char *name, int len, int xattr);
 187 int  smbfs_smb_rename(struct smbnode *src, struct smbnode *tdnp,
 188         const char *tname, int tnmlen, struct smb_cred *scrp);
 189 int  smbfs_smb_t2rename(struct smbnode *np, struct smbnode *tdnp,
 190         const char *tname, int tnmlen, struct smb_cred *scrp, int overwrite);
 191 int  smbfs_smb_move(struct smbnode *src, struct smbnode *tdnp,
 192         const char *tname, int tnmlen, uint16_t flags, struct smb_cred *scrp);
 193 int  smbfs_smb_mkdir(struct smbnode *dnp, const char *name, int len,
 194         struct smb_cred *scrp);
 195 int  smbfs_smb_rmdir(struct smbnode *np, struct smb_cred *scrp);
 196 int  smbfs_smb_findopen(struct smbnode *dnp, const char *wildcard, int wclen,
 197         int attr, struct smb_cred *scrp, struct smbfs_fctx **ctxpp);
 198 int  smbfs_smb_findnext(struct smbfs_fctx *ctx, int limit,
 199         struct smb_cred *scrp);
 200 int  smbfs_smb_findclose(struct smbfs_fctx *ctx, struct smb_cred *scrp);
 201 int  smbfs_fullpath(struct mbchain *mbp, struct smb_vc *vcp,
 202         struct smbnode *dnp, const char *name, int nmlen, uint8_t sep);
 203 int  smbfs_smb_lookup(struct smbnode *dnp, const char **namep, int *nmlenp,
 204         struct smbfattr *fap, struct smb_cred *scrp);
 205 int  smbfs_smb_hideit(struct smbnode *np, const char *name, int len,
 206         struct smb_cred *scrp);
 207 int  smbfs_smb_unhideit(struct smbnode *np, const char *name, int len,
 208                         struct smb_cred *scrp);
```



```
 209 int smbfs_smb_flush(struct smbnode *np, struct smb_cred *scrp);
 210 int smbfs_0extend(vnode_t *vp, uint16_t fid, len_t from, len_t to,
 211                 struct smb_cred *scredp, int timo);

 213 /* get/set security descriptor */
 214 int  smbfs_smb_getsec_m(struct smb_share *ssp, uint16_t fid,
 215         struct smb_cred *scrp, uint32_t selector,
 216         mblk_t **res, uint32_t *reslen);
 217 int  smbfs_smb_setsec_m(struct smb_share *ssp, uint16_t fid,
 218         struct smb_cred *scrp, uint32_t selector, mblk_t **mp);

 220 /*
 221  * VFS-level init, fini stuff
 222  */

 224 int smbfs_vfsinit(void);
 225 void smbfs_vfsfini(void);
 226 int smbfs_subrinit(void);
 227 void smbfs_subrfini(void);
 228 int smbfs_clntinit(void);
 229 void smbfs_clntfini(void);

 231 void smbfs_zonelist_add(smbmntinfo_t *smi);
 232 void smbfs_zonelist_remove(smbmntinfo_t *smi);

 234 int smbfs_check_table(struct vfs *vfsp, struct smbnode *srp);
 235 void smbfs_destroy_table(struct vfs *vfsp);
 236 void smbfs_rflush(struct vfs *vfsp, cred_t *cr);

 238 /*
 239  * Function definitions - those having to do with
 240  * smbfs nodes, vnodes, etc
 241  */

 243 void smbfs_attrcache_prune(struct smbnode *np);
 244 void smbfs_attrcache_remove(struct smbnode *np);
 245 void smbfs_attrcache_rm_locked(struct smbnode *np);
 246 #ifndef DEBUG
 247 #define smbfs_attrcache_rm_locked(np)   (np)->r_attrtime = gethrtime()
 248 #endif
 249 void smbfs_attr_touchdir(struct smbnode *dnp);
 250 void smbfs_attrcache_fa(vnode_t *vp, struct smbfattr *fap);
 251 void smbfs_cache_check(struct vnode *vp, struct smbfattr *fap);

 253 void smbfs_addfree(struct smbnode *sp);
 254 void smbfs_rmhash(struct smbnode *);

 256 void smbfs_invalidate_pages(vnode_t *vp, u_offset_t off, cred_t *cr);

 258 #endif /* ! codereview */
 259 /* See avl_create in smbfs_vfsops.c */
 260 void smbfs_init_hash_avl(avl_tree_t *);

 262 uint32_t smbfs_gethash(const char *rpath, int prlen);
 263 uint32_t smbfs_getino(struct smbnode *dnp, const char *name, int nmlen);

 265 extern struct smbfattr smbfs_fattr0;
 266 smbnode_t *smbfs_node_findcreate(smbmntinfo_t *mi,
 267     const char *dir, int dirlen,
 268     const char *name, int nmlen,
 269     char sep, struct smbfattr *fap);

 271 int smbfs_nget(vnode_t *dvp, const char *name, int nmlen,
 272         struct smbfattr *fap, vnode_t **vpp);

 274 void smbfs_fname_tolocal(struct smbfs_fctx *ctx);
```

```
275 char    *smbfs_name_alloc(const char *name, int nmlen);
276 void    smbfs_name_free(const char *name, int nmlen);

278 int smbfs_readvnode(vnode_t *, uio_t *, cred_t *, struct vattr *);
279 int smbfs_writevnode(vnode_t *vp, uio_t *uiop, cred_t *cr,
280                      int ioflag, int timo);
281 int smbfsgetattr(vnode_t *vp, struct vattr *vap, cred_t *cr);

283 /* smbfs ACL support */
284 int smbfs_acl_getids(vnode_t *, cred_t *);
285 int smbfs_acl_setids(vnode_t *, vattr_t *, cred_t *);
286 int smbfs_acl_getvsa(vnode_t *, vsecattr_t *, int, cred_t *);
287 int smbfs_acl_setvsa(vnode_t *, vsecattr_t *, int, cred_t *);
288 int smbfs_acl_iocget(vnode_t *, intptr_t, int, cred_t *);
289 int smbfs_acl_iocset(vnode_t *, intptr_t, int, cred_t *);

291 /* smbfs_xattr.c */
292 int smbfs_get_xattrdir(vnode_t *dvp, vnode_t **vpp, cred_t *cr, int);
293 int smbfs_xa_parent(vnode_t *vp, vnode_t **vpp);
294 int smbfs_xa_exists(vnode_t *vp, cred_t *cr);
295 int smbfs_xa_getfattr(struct smbnode *np, struct smbfattr *fap,
296         struct smb_cred *scrp);
297 int smbfs_xa_findopen(struct smbfs_fctx *ctx, struct smbnode *dnp,
298         const char *name, int nmlen);
299 int smbfs_xa_findnext(struct smbfs_fctx *ctx, uint16_t limit);
300 int smbfs_xa_findclose(struct smbfs_fctx *ctx);

302 /* For Solaris, interruptible rwlock */
303 int smbfs_rw_enter_sig(smbfs_rwlock_t *l, krw_t rw, int intr);
304 int smbfs_rw_tryenter(smbfs_rwlock_t *l, krw_t rw);
305 void smbfs_rw_exit(smbfs_rwlock_t *l);
306 int smbfs_rw_lock_held(smbfs_rwlock_t *l, krw_t rw);
307 void smbfs_rw_init(smbfs_rwlock_t *l, char *name, krw_type_t type, void *arg);
308 void smbfs_rw_destroy(smbfs_rwlock_t *l);

310 #endif /* !_FS_SMBFS_SMBFS_SUBR_H_ */
```

```
**********************************************************
   29801 Sat Aug 18 10:48:43 2012
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_subr2.c
*** NO COMMENTS ***
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  *
  25  *      Copyright (c) 1983,1984,1985,1986,1987,1988,1989  AT&T.
  26  *      All rights reserved.
  27  */

  29 /*
  30  * Node hash implementation initially borrowed from NFS (nfs_subr.c)
  31  * but then heavily modified. It's no longer an array of hash lists,
  32  * but an AVL tree per mount point.  More on this below.
  33  */

  35 #include <sys/param.h>
  36 #include <sys/systm.h>
  37 #include <sys/time.h>
  38 #include <sys/vnode.h>
  39 #include <sys/bitmap.h>
  40 #include <sys/dnlc.h>
  41 #include <sys/kmem.h>
  42 #include <sys/sunddi.h>
  43 #include <sys/sysmacros.h>

  45 #include <netsmb/smb_osdep.h>

  47 #include <netsmb/smb.h>
  48 #include <netsmb/smb_conn.h>
  49 #include <netsmb/smb_subr.h>
  50 #include <netsmb/smb_rq.h>

  52 #include <smbfs/smbfs.h>
  53 #include <smbfs/smbfs_node.h>
  54 #include <smbfs/smbfs_subr.h>

  56 /*
  57  * The AVL trees (now per-mount) allow finding an smbfs node by its
  58  * full remote path name.  It also allows easy traversal of all nodes
  59  * below (path wise) any given node.  A reader/writer lock for each
  60  * (per mount) AVL tree is used to control access and to synchronize
  61  * lookups, additions, and deletions from that AVL tree.
```



```
  62  *
  63  * Previously, this code use a global array of hash chains, each with
  64  * its own rwlock.  A few struct members, functions, and comments may
  65  * still refer to a "hash", and those should all now be considered to
  66  * refer to the per-mount AVL tree that replaced the old hash chains.
  67  * (i.e. member smi_hash_lk, function sn_hashfind, etc.)
  68  *
  69  * The smbnode freelist is organized as a doubly linked list with
  70  * a head pointer.  Additions and deletions are synchronized via
  71  * a single mutex.
  72  *
  73  * In order to add an smbnode to the free list, it must be linked into
  74  * the mount's AVL tree and the exclusive lock for the AVL must be held.
  75  * If an smbnode is not linked into the AVL tree, then it is destroyed
  76  * because it represents no valuable information that can be reused
  77  * about the file.  The exclusive lock for the AVL tree must be held
  78  * in order to prevent a lookup in the AVL tree from finding the
  79  * smbnode and using it and assuming that the smbnode is not on the
  80  * freelist.  The lookup in the AVL tree will have the AVL tree lock
  81  * held, either exclusive or shared.
  82  *
  83  * The vnode reference count for each smbnode is not allowed to drop
  84  * below 1.  This prevents external entities, such as the VM
  85  * subsystem, from acquiring references to vnodes already on the
  86  * freelist and then trying to place them back on the freelist
  87  * when their reference is released.  This means that the when an
  88  * smbnode is looked up in the AVL tree, then either the smbnode
  89  * is removed from the freelist and that reference is tranfered to
  90  * the new reference or the vnode reference count must be incremented
  91  * accordingly.  The mutex for the freelist must be held in order to
  92  * accurately test to see if the smbnode is on the freelist or not.
  93  * The AVL tree lock might be held shared and it is possible that
  94  * two different threads may race to remove the smbnode from the
  95  * freelist.  This race can be resolved by holding the mutex for the
  96  * freelist.  Please note that the mutex for the freelist does not
  97  * need to held if the smbnode is not on the freelist.  It can not be
  98  * placed on the freelist due to the requirement that the thread
  99  * putting the smbnode on the freelist must hold the exclusive lock
 100  * for the AVL tree and the thread doing the lookup in the AVL tree
 101  * is holding either a shared or exclusive lock for the AVL tree.
 102  *
 103  * The lock ordering is:
 104  *
 105  *      AVL tree lock -> vnode lock
 106  *      AVL tree lock -> freelist lock
 107  */

 109 static kmutex_t smbfreelist_lock;
 110 static smbnode_t *smbfreelist = NULL;
 111 static ulong_t  smbnodenew = 0;
 112 long    nsmbnode = 0;

 114 static struct kmem_cache *smbnode_cache;

 116 static const vsecattr_t smbfs_vsa0 = { 0 };

 118 /*
 119  * Mutex to protect the following variables:
 120  *      smbfs_major
 121  *      smbfs_minor
 122  */
 123 kmutex_t smbfs_minor_lock;
 124 int smbfs_major;
 125 int smbfs_minor;

 127 /* See smbfs_node_findcreate() */
```

```
128 struct smbfattr smbfs_fattr0;

130 /*
131  * Local functions.
132  * SN for Smb Node
133  */
134 static void sn_rmfree(smbnode_t *);
135 static void sn_inactive(smbnode_t *);
136 static void sn_addhash_locked(smbnode_t *, avl_index_t);
137 static void sn_rmhash_locked(smbnode_t *);
138 static void sn_destroy_node(smbnode_t *);
139 void smbfs_kmem_reclaim(void *cdrarg);

141 static smbnode_t *
142 sn_hashfind(smbmntinfo_t *, const char *, int, avl_index_t *);

144 static smbnode_t *
145 make_smbnode(smbmntinfo_t *, const char *, int, int *);

147 /*
148  * Free the resources associated with an smbnode.
149  * Note: This is different from smbfs_inactive
150  *
151  * NFS: nfs_subr.c:rinactive
152  */
153 static void
154 sn_inactive(smbnode_t *np)
155 {
156         vsecattr_t      ovsa;
157         cred_t          *oldcr;
158         char            *orpath;
159         int             orplen;
160         vnode_t         *vp;
161 #endif /* ! codereview */

163         /*
164          * Flush and invalidate all pages
160          * Flush and invalidate all pages (todo)
165          * Free any held credentials and caches...
166          * etc.  (See NFS code)
167          */
168         mutex_enter(&np->r_statelock);

170         ovsa = np->r_secattr;
171         np->r_secattr = smbfs_vsa0;
172         np->r_sectime = 0;

174         oldcr = np->r_cred;
175         np->r_cred = NULL;

177         orpath = np->n_rpath;
178         orplen = np->n_rplen;
179         np->n_rpath = NULL;
180         np->n_rplen = 0;

182         mutex_exit(&np->r_statelock);

184         vp = SMBTOV(np);
185         if (vn_has_cached_data(vp)) {
186                 smbfs_invalidate_pages(vp, (u_offset_t) 0, oldcr);
187         }

189 #endif /* ! codereview */
190         if (ovsa.vsa_aclentp != NULL)
191                 kmem_free(ovsa.vsa_aclentp, ovsa.vsa_aclentsz);
```



```
193         if (oldcr != NULL)
194                 crfree(oldcr);

196         if (orpath != NULL)
197                 kmem_free(orpath, orplen + 1);
198 }

200 /*
201  * Find and optionally create an smbnode for the passed
202  * mountinfo, directory, separator, and name.  If the
203  * desired smbnode already exists, return a reference.
204  * If the file attributes pointer is non-null, the node
205  * is created if necessary and linked into the AVL tree.
206  *
207  * Callers that need a node created but don't have the
208  * real attributes pass smbfs_fattr0 to force creation.
209  *
210  * Note: make_smbnode() may upgrade the "hash" lock to exclusive.
211  *
212  * NFS: nfs_subr.c:makenfsnode
213  */
214 smbnode_t *
215 smbfs_node_findcreate(
216         smbmntinfo_t *mi,
217         const char *dirnm,
218         int dirlen,
219         const char *name,
220         int nmlen,
221         char sep,
222         struct smbfattr *fap)
223 {
224         char tmpbuf[256];
225         size_t rpalloc;
226         char *p, *rpath;
227         int rplen;
228         smbnode_t *np;
229         vnode_t *vp;
230         int newnode;

232         /*
233          * Build the search string, either in tmpbuf or
234          * in allocated memory if larger than tmpbuf.
235          */
236         rplen = dirlen;
237         if (sep != '\0')
238                 rplen++;
239         rplen += nmlen;
240         if (rplen < sizeof (tmpbuf)) {
241                 /* use tmpbuf */
242                 rpalloc = 0;
243                 rpath = tmpbuf;
244         } else {
245                 rpalloc = rplen + 1;
246                 rpath = kmem_alloc(rpalloc, KM_SLEEP);
247         }
248         p = rpath;
249         bcopy(dirnm, p, dirlen);
250         p += dirlen;
251         if (sep != '\0')
252                 *p++ = sep;
253         if (name != NULL) {
254                 bcopy(name, p, nmlen);
255                 p += nmlen;
256         }
257         ASSERT(p == rpath + rplen);
```

```
259         /*
260          * Find or create a node with this path.
261          */
262         rw_enter(&mi->smi_hash_lk, RW_READER);
263         if (fap == NULL)
264                 np = sn_hashfind(mi, rpath, rplen, NULL);
265         else
266                 np = make_smbnode(mi, rpath, rplen, &newnode);
267         rw_exit(&mi->smi_hash_lk);

269         if (rpalloc)
270                 kmem_free(rpath, rpalloc);

272         if (fap == NULL) {
273                 /*
274                  * Caller is "just looking" (no create)
275                  * so np may or may not be NULL here.
276                  * Either way, we're done.
277                  */
278                 return (np);
279         }

281         /*
282          * We should have a node, possibly created.
283          * Do we have (real) attributes to apply?
284          */
285         ASSERT(np != NULL);
286         if (fap == &smbfs_fattr0)
287                 return (np);

289         /*
290          * Apply the given attributes to this node,
291          * dealing with any cache impact, etc.
292          */
293         vp = SMBTOV(np);
294         if (!newnode) {
295                 /*
296                  * Found an existing node.
297                  * Maybe purge caches...
298                  */
299                 smbfs_cache_check(vp, fap);
300         }
301         smbfs_attrcache_fa(vp, fap);

303         /*
304          * Note NFS sets vp->v_type here, assuming it
305          * can never change for the life of a node.
306          * We allow v_type to change, and set it in
307          * smbfs_attrcache().  Also: mode, uid, gid
308          */
309         return (np);
310 }

312 /*
313  * NFS: nfs_subr.c:rtablehash
314  * We use smbfs_hash().
315  */

317 /*
318  * Find or create an smbnode.
319  * NFS: nfs_subr.c:make_rnode
320  */
321 static smbnode_t *
322 make_smbnode(
323         smbmntinfo_t *mi,
324         const char *rpath,
```



```
325         int rplen,
326         int *newnode)
327 {
328         smbnode_t *np;
329         smbnode_t *tnp;
330         vnode_t *vp;
331         vfs_t *vfsp;
332         avl_index_t where;
333         char *new_rpath = NULL;

335         ASSERT(RW_READ_HELD(&mi->smi_hash_lk));
336         vfsp = mi->smi_vfsp;

338 start:
339         np = sn_hashfind(mi, rpath, rplen, NULL);
340         if (np != NULL) {
341                 *newnode = 0;
342                 return (np);
343         }

345         /* Note: will retake this lock below. */
346         rw_exit(&mi->smi_hash_lk);

348         /*
349          * see if we can find something on the freelist
350          */
351         mutex_enter(&smbfreelist_lock);
352         if (smbfreelist != NULL && smbnodenew >= nsmbnode) {
353                 np = smbfreelist;
354                 sn_rmfree(np);
355                 mutex_exit(&smbfreelist_lock);

357                 vp = SMBTOV(np);

359                 if (np->r_flags & RHASHED) {
360                         smbmntinfo_t *tmp_mi = np->n_mount;
361                         ASSERT(tmp_mi != NULL);
362                         rw_enter(&tmp_mi->smi_hash_lk, RW_WRITER);
363                         mutex_enter(&vp->v_lock);
364                         if (vp->v_count > 1) {
365                                 vp->v_count--;
366                                 mutex_exit(&vp->v_lock);
367                                 rw_exit(&tmp_mi->smi_hash_lk);
368                                 /* start over */
369                                 rw_enter(&mi->smi_hash_lk, RW_READER);
370                                 goto start;
371                         }
372                         mutex_exit(&vp->v_lock);
373                         sn_rmhash_locked(np);
374                         rw_exit(&tmp_mi->smi_hash_lk);
375                 }

377                 sn_inactive(np);

379                 mutex_enter(&vp->v_lock);
380                 if (vp->v_count > 1) {
381                         vp->v_count--;
382                         mutex_exit(&vp->v_lock);
383                         rw_enter(&mi->smi_hash_lk, RW_READER);
384                         goto start;
385                 }
386                 mutex_exit(&vp->v_lock);
387                 vn_invalid(vp);
388                 /*
389                  * destroy old locks before bzero'ing and
390                  * recreating the locks below.
```

```
391                  */
392                 smbfs_rw_destroy(&np->r_rwlock);
393                 smbfs_rw_destroy(&np->r_lkserlock);
394                 mutex_destroy(&np->r_statelock);
395                 cv_destroy(&np->r_cv);
396                 /*
397                  * Make sure that if smbnode is recycled then
398                  * VFS count is decremented properly before
399                  * reuse.
400                  */
401                 VFS_RELE(vp->v_vfsp);
402                 vn_reinit(vp);
403         } else {
404                 /*
405                  * allocate and initialize a new smbnode
406                  */
407                 vnode_t *new_vp;

409                 mutex_exit(&smbfreelist_lock);

411                 np = kmem_cache_alloc(smbnode_cache, KM_SLEEP);
412                 new_vp = vn_alloc(KM_SLEEP);

414                 atomic_add_long((ulong_t *)&smbnodenew, 1);
415                 vp = new_vp;
416         }

418         /*
419          * Allocate and copy the rpath we'll need below.
420          */
421         new_rpath = kmem_alloc(rplen + 1, KM_SLEEP);
422         bcopy(rpath, new_rpath, rplen);
423         new_rpath[rplen] = '\0';

425         /* Initialize smbnode_t */
426         bzero(np, sizeof (*np));

428         smbfs_rw_init(&np->r_rwlock, NULL, RW_DEFAULT, NULL);
429         smbfs_rw_init(&np->r_lkserlock, NULL, RW_DEFAULT, NULL);
430         mutex_init(&np->r_statelock, NULL, MUTEX_DEFAULT, NULL);
431         cv_init(&np->r_cv, NULL, CV_DEFAULT, NULL);
432         /* cv_init(&np->r_commit.c_cv, NULL, CV_DEFAULT, NULL); */

434         np->r_vnode = vp;
435         np->n_mount = mi;

437         np->n_fid = SMB_FID_UNUSED;
438         np->n_uid = mi->smi_uid;
439         np->n_gid = mi->smi_gid;
440         /* Leave attributes "stale." */

442 #if 0 /* XXX dircache */
443         /*
444          * We don't know if it's a directory yet.
445          * Let the caller do this?  XXX
446          */
447         avl_create(&np->r_dir, compar, sizeof (rddir_cache),
448             offsetof(rddir_cache, tree));
449 #endif

451         /* Now fill in the vnode. */
452         vn_setops(vp, smbfs_vnodeops);
453         vp->v_data = (caddr_t)np;
454         VFS_HOLD(vfsp);
455         vp->v_vfsp = vfsp;
456         vp->v_type = VNON;
```



```
458         /*
459          * We entered with mi->smi_hash_lk held (reader).
460          * Retake it now, (as the writer).
461          * Will return with it held.
462          */
463         rw_enter(&mi->smi_hash_lk, RW_WRITER);

465         /*
466          * There is a race condition where someone else
467          * may alloc the smbnode while no locks are held,
468          * so check again and recover if found.
469          */
470         tnp = sn_hashfind(mi, rpath, rplen, &where);
471         if (tnp != NULL) {
472                 /*
473                  * Lost the race.  Put the node we were building
474                  * on the free list and return the one we found.
475                  */
476                 rw_exit(&mi->smi_hash_lk);
477                 kmem_free(new_rpath, rplen + 1);
478                 smbfs_addfree(np);
479                 rw_enter(&mi->smi_hash_lk, RW_READER);
480                 *newnode = 0;
481                 return (tnp);
482         }

484         /*
485          * Hash search identifies nodes by the remote path
486          * (n_rpath) so fill that in now, before linking
487          * this node into the node cache (AVL tree).
488          */
489         np->n_rpath = new_rpath;
490         np->n_rplen = rplen;
491         np->n_ino = smbfs_gethash(new_rpath, rplen);

493         sn_addhash_locked(np, where);
494         *newnode = 1;
495         return (np);
496 }

498 /*
499  * smbfs_addfree
500  * Put an smbnode on the free list, or destroy it immediately
501  * if it offers no value were it to be reclaimed later.  Also
502  * destroy immediately when we have too many smbnodes, etc.
503  *
504  * Normally called by smbfs_inactive, but also
505  * called in here during cleanup operations.
506  *
507  * NFS: nfs_subr.c:rp_addfree
508  */
509 void
510 smbfs_addfree(smbnode_t *np)
511 {
512         vnode_t *vp;
513         struct vfs *vfsp;
514         smbmntinfo_t *mi;

516         ASSERT(np->r_freef == NULL && np->r_freeb == NULL);

518         vp = SMBTOV(np);
519         ASSERT(vp->v_count >= 1);

521         vfsp = vp->v_vfsp;
522         mi = VFTOSMI(vfsp);
```

```
524         /*
525          * If there are no more references to this smbnode and:
526          * we have too many smbnodes allocated, or if the node
527          * is no longer accessible via the AVL tree (!RHASHED),
528          * or an i/o error occurred while writing to the file,
529          * or it's part of an unmounted FS, then try to destroy
530          * it instead of putting it on the smbnode freelist.
531          */
532         if (np->r_count == 0 && (
533             (np->r_flags & RHASHED) == 0 ||
534             (np->r_error != 0) ||
535             (vfsp->vfs_flag & VFS_UNMOUNTED) ||
536             (smbnodenew > nsmbnode))) {

538                 /* Try to destroy this node. */

540                 if (np->r_flags & RHASHED) {
541                         rw_enter(&mi->smi_hash_lk, RW_WRITER);
542                         mutex_enter(&vp->v_lock);
543                         if (vp->v_count > 1) {
544                                 vp->v_count--;
545                                 mutex_exit(&vp->v_lock);
546                                 rw_exit(&mi->smi_hash_lk);
547                                 return;
548                                 /*
549                                  * Will get another call later,
550                                  * via smbfs_inactive.
551                                  */
552                         }
553                         mutex_exit(&vp->v_lock);
554                         sn_rmhash_locked(np);
555                         rw_exit(&mi->smi_hash_lk);
556                 }

558                 sn_inactive(np);

560                 /*
561                  * Recheck the vnode reference count.  We need to
562                  * make sure that another reference has not been
563                  * acquired while we were not holding v_lock.  The
564                  * smbnode is not in the smbnode "hash" AVL tree, so
565                  * the only way for a reference to have been acquired
566                  * is for a VOP_PUTPAGE because the smbnode was marked
567                  * with RDIRTY or for a modified page.  This vnode
568                  * reference may have been acquired before our call
569                  * to sn_inactive.  The i/o may have been completed,
570                  * thus allowing sn_inactive to complete, but the
571                  * reference to the vnode may not have been released
572                  * yet.  In any case, the smbnode can not be destroyed
573                  * until the other references to this vnode have been
574                  * released.  The other references will take care of
575                  * either destroying the smbnode or placing it on the
576                  * smbnode freelist.  If there are no other references,
577                  * then the smbnode may be safely destroyed.
578                  */
579                 mutex_enter(&vp->v_lock);
580                 if (vp->v_count > 1) {
581                         vp->v_count--;
582                         mutex_exit(&vp->v_lock);
583                         return;
584                 }
585                 mutex_exit(&vp->v_lock);

587                 sn_destroy_node(np);
588                 return;
```



```
589         }

591         /*
592          * Lock the AVL tree and then recheck the reference count
593          * to ensure that no other threads have acquired a reference
594          * to indicate that the smbnode should not be placed on the
595          * freelist.  If another reference has been acquired, then
596          * just release this one and let the other thread complete
597          * the processing of adding this smbnode to the freelist.
598          */
599         rw_enter(&mi->smi_hash_lk, RW_WRITER);

601         mutex_enter(&vp->v_lock);
602         if (vp->v_count > 1) {
603                 vp->v_count--;
604                 mutex_exit(&vp->v_lock);
605                 rw_exit(&mi->smi_hash_lk);
606                 return;
607         }
608         mutex_exit(&vp->v_lock);

610         /*
611          * Put this node on the free list.
612          */
613         mutex_enter(&smbfreelist_lock);
614         if (smbfreelist == NULL) {
615                 np->r_freef = np;
616                 np->r_freeb = np;
617                 smbfreelist = np;
618         } else {
619                 np->r_freef = smbfreelist;
620                 np->r_freeb = smbfreelist->r_freeb;
621                 smbfreelist->r_freeb->r_freef = np;
622                 smbfreelist->r_freeb = np;
623         }
624         mutex_exit(&smbfreelist_lock);

626         rw_exit(&mi->smi_hash_lk);
627 }

629 /*
630  * Remove an smbnode from the free list.
631  *
632  * The caller must be holding smbfreelist_lock and the smbnode
633  * must be on the freelist.
634  *
635  * NFS: nfs_subr.c:rp_rmfree
636  */
637 static void
638 sn_rmfree(smbnode_t *np)
639 {

641         ASSERT(MUTEX_HELD(&smbfreelist_lock));
642         ASSERT(np->r_freef != NULL && np->r_freeb != NULL);

644         if (np == smbfreelist) {
645                 smbfreelist = np->r_freef;
646                 if (np == smbfreelist)
647                         smbfreelist = NULL;
648         }

650         np->r_freeb->r_freef = np->r_freef;
651         np->r_freef->r_freeb = np->r_freeb;

653         np->r_freef = np->r_freeb = NULL;
654 }
```

```
656 /*
657  * Put an smbnode in the "hash" AVL tree.
658  *
659  * The caller must be hold the rwlock as writer.
660  *
661  * NFS: nfs_subr.c:rp_addhash
662  */
663 static void
664 sn_addhash_locked(smbnode_t *np, avl_index_t where)
665 {
666         smbmntinfo_t *mi = np->n_mount;

668         ASSERT(RW_WRITE_HELD(&mi->smi_hash_lk));
669         ASSERT(!(np->r_flags & RHASHED));

671         avl_insert(&mi->smi_hash_avl, np, where);

673         mutex_enter(&np->r_statelock);
674         np->r_flags |= RHASHED;
675         mutex_exit(&np->r_statelock);
676 }

678 /*
679  * Remove an smbnode from the "hash" AVL tree.
680  *
681  * The caller must hold the rwlock as writer.
682  *
683  * NFS: nfs_subr.c:rp_rmhash_locked
684  */
685 static void
686 sn_rmhash_locked(smbnode_t *np)
687 {
688         smbmntinfo_t *mi = np->n_mount;

690         ASSERT(RW_WRITE_HELD(&mi->smi_hash_lk));
691         ASSERT(np->r_flags & RHASHED);

693         avl_remove(&mi->smi_hash_avl, np);

695         mutex_enter(&np->r_statelock);
696         np->r_flags &= ~RHASHED;
697         mutex_exit(&np->r_statelock);
698 }

700 /*
701  * Remove an smbnode from the "hash" AVL tree.
702  *
703  * The caller must not be holding the rwlock.
704  */
705 void
706 smbfs_rmhash(smbnode_t *np)
707 {
708         smbmntinfo_t *mi = np->n_mount;

710         rw_enter(&mi->smi_hash_lk, RW_WRITER);
711         sn_rmhash_locked(np);
712         rw_exit(&mi->smi_hash_lk);
713 }

715 /*
716  * Lookup an smbnode by remote pathname
717  *
718  * The caller must be holding the AVL rwlock, either shared or exclusive.
719  *
720  * NFS: nfs_subr.c:rfind
```



```
721  */
722 static smbnode_t *
723 sn_hashfind(
724         smbmntinfo_t *mi,
725         const char *rpath,
726         int rplen,
727         avl_index_t *pwhere) /* optional */
728 {
729         smbfs_node_hdr_t nhdr;
730         smbnode_t *np;
731         vnode_t *vp;

733         ASSERT(RW_LOCK_HELD(&mi->smi_hash_lk));

735         bzero(&nhdr, sizeof (nhdr));
736         nhdr.hdr_n_rpath = (char *)rpath;
737         nhdr.hdr_n_rplen = rplen;

739         /* See smbfs_node_cmp below. */
740         np = avl_find(&mi->smi_hash_avl, &nhdr, pwhere);

742         if (np == NULL)
743                 return (NULL);

745         /*
746          * Found it in the "hash" AVL tree.
747          * Remove from free list, if necessary.
748          */
749         vp = SMBTOV(np);
750         if (np->r_freef != NULL) {
751                 mutex_enter(&smbfreelist_lock);
752                 /*
753                  * If the smbnode is on the freelist,
754                  * then remove it and use that reference
755                  * as the new reference.  Otherwise,
756                  * need to increment the reference count.
757                  */
758                 if (np->r_freef != NULL) {
759                         sn_rmfree(np);
760                         mutex_exit(&smbfreelist_lock);
761                 } else {
762                         mutex_exit(&smbfreelist_lock);
763                         VN_HOLD(vp);
764                 }
765         } else
766                 VN_HOLD(vp);

768         return (np);
769 }

771 static int
772 smbfs_node_cmp(const void *va, const void *vb)
773 {
774         const smbfs_node_hdr_t *a = va;
775         const smbfs_node_hdr_t *b = vb;
776         int clen, diff;

778         /*
779          * Same semantics as strcmp, but does not
780          * assume the strings are null terminated.
781          */
782         clen = (a->hdr_n_rplen < b->hdr_n_rplen) ?
783             a->hdr_n_rplen : b->hdr_n_rplen;
784         diff = strncmp(a->hdr_n_rpath, b->hdr_n_rpath, clen);
785         if (diff < 0)
786                 return (-1);
```

```
787         if (diff > 0)
788                 return (1);
789         /* they match through clen */
790         if (b->hdr_n_rplen > clen)
791                 return (-1);
792         if (a->hdr_n_rplen > clen)
793                 return (1);
794         return (0);
795 }

797 /*
798  * Setup the "hash" AVL tree used for our node cache.
799  * See: smbfs_mount, smbfs_destroy_table.
800  */
801 void
802 smbfs_init_hash_avl(avl_tree_t *avl)
803 {
804         avl_create(avl, smbfs_node_cmp, sizeof (smbnode_t),
805             offsetof(smbnode_t, r_avl_node));
806 }

808 /*
809  * Invalidate the cached attributes for all nodes "under" the
810  * passed-in node.  Note: the passed-in node is NOT affected by
811  * this call.  This is used both for files under some directory
812  * after the directory is deleted or renamed, and for extended
813  * attribute files (named streams) under a plain file after that
814  * file is renamed or deleted.
815  *
816  * Do this by walking the AVL tree starting at the passed in node,
817  * and continuing while the visited nodes have a path prefix matching
818  * the entire path of the passed-in node, and a separator just after
819  * that matching path prefix.  Watch out for cases where the AVL tree
820  * order may not exactly match the order of an FS walk, i.e.
821  * consider this sequence:
822  *      "foo"           (directory)
823  *      "foo bar"       (name containing a space)
824  *      "foo/bar"
825  * The walk needs to skip "foo bar" and keep going until it finds
826  * something that doesn't match the "foo" name prefix.
827  */
828 void
829 smbfs_attrcache_prune(smbnode_t *top_np)
830 {
831         smbmntinfo_t *mi;
832         smbnode_t *np;
833         char *rpath;
834         int rplen;

836         mi = top_np->n_mount;
837         rw_enter(&mi->smi_hash_lk, RW_READER);

839         np = top_np;
840         rpath = top_np->n_rpath;
841         rplen = top_np->n_rplen;
842         for (;;) {
843                 np = avl_walk(&mi->smi_hash_avl, np, AVL_AFTER);
844                 if (np == NULL)
845                         break;
846                 if (np->n_rplen < rplen)
847                         break;
848                 if (0 != strncmp(np->n_rpath, rpath, rplen))
849                         break;
850                 if (np->n_rplen > rplen && (
851                     np->n_rpath[rplen] == ':' ||
852                     np->n_rpath[rplen] == '\\'))
```



```
853                         smbfs_attrcache_remove(np);
854         }

856         rw_exit(&mi->smi_hash_lk);
857 }

859 #ifdef SMB_VNODE_DEBUG
860 int smbfs_check_table_debug = 1;
861 #else /* SMB_VNODE_DEBUG */
862 int smbfs_check_table_debug = 0;
863 #endif /* SMB_VNODE_DEBUG */


866 /*
867  * Return 1 if there is a active vnode belonging to this vfs in the
868  * smbnode cache.
869  *
870  * Several of these checks are done without holding the usual
871  * locks.  This is safe because destroy_smbtable(), smbfs_addfree(),
872  * etc. will redo the necessary checks before actually destroying
873  * any smbnodes.
874  *
875  * NFS: nfs_subr.c:check_rtable
876  *
877  * Debugging changes here relative to NFS.
878  * Relatively harmless, so left 'em in.
879  */
880 int
881 smbfs_check_table(struct vfs *vfsp, smbnode_t *rtnp)
882 {
883         smbmntinfo_t *mi;
884         smbnode_t *np;
885         vnode_t *vp;
886         int busycnt = 0;

888         mi = VFTOSMI(vfsp);
889         rw_enter(&mi->smi_hash_lk, RW_READER);
890         for (np = avl_first(&mi->smi_hash_avl); np != NULL;
891             np = avl_walk(&mi->smi_hash_avl, np, AVL_AFTER)) {

893                 if (np == rtnp)
894                         continue; /* skip the root */
895                 vp = SMBTOV(np);

897                 /* Now the 'busy' checks: */
898                 /* Not on the free list? */
899                 if (np->r_freef == NULL) {
900                         SMBVDEBUG("!r_freef: node=0x%p, rpath=%s\n",
901                             (void *)np, np->n_rpath);
902                         busycnt++;
903                 }

905                 /* Has dirty pages? */
906                 if (vn_has_cached_data(vp) &&
907                     (np->r_flags & RDIRTY)) {
908                         SMBVDEBUG("is dirty: node=0x%p, rpath=%s\n",
909                             (void *)np, np->n_rpath);
910                         busycnt++;
911                 }

913                 /* Other refs? (not reflected in v_count) */
914                 if (np->r_count > 0) {
915                         SMBVDEBUG("+r_count: node=0x%p, rpath=%s\n",
916                             (void *)np, np->n_rpath);
917                         busycnt++;
918                 }
```

```
920                 if (busycnt && !smbfs_check_table_debug)
921                         break;
922
923         }
924         rw_exit(&mi->smi_hash_lk);
925
926         return (busycnt);
927 }
928
929 /*
930  * Destroy inactive vnodes from the AVL tree which belong to this
931  * vfs.  It is essential that we destroy all inactive vnodes during a
932  * forced unmount as well as during a normal unmount.
933  *
934  * NFS: nfs_subr.c:destroy_rtable
935  *
936  * In here, we're normally destrying all or most of the AVL tree,
937  * so the natural choice is to use avl_destroy_nodes.  However,
938  * there may be a few busy nodes that should remain in the AVL
939  * tree when we're done.  The solution: use a temporary tree to
940  * hold the busy nodes until we're done destroying the old tree,
941  * then copy the temporary tree over the (now emtpy) real tree.
942  */
943 void
944 smbfs_destroy_table(struct vfs *vfsp)
945 {
946         avl_tree_t tmp_avl;
947         smbmntinfo_t *mi;
948         smbnode_t *np;
949         smbnode_t *rlist;
950         void *v;
951
952         mi = VFTOSMI(vfsp);
953         rlist = NULL;
954         smbfs_init_hash_avl(&tmp_avl);
955
956         rw_enter(&mi->smi_hash_lk, RW_WRITER);
957         v = NULL;
958         while ((np = avl_destroy_nodes(&mi->smi_hash_avl, &v)) != NULL) {
959
960                 mutex_enter(&smbfreelist_lock);
961                 if (np->r_freef == NULL) {
962                         /*
963                          * Busy node (not on the free list).
964                          * Will keep in the final AVL tree.
965                          */
966                         mutex_exit(&smbfreelist_lock);
967                         avl_add(&tmp_avl, np);
968                 } else {
969                         /*
970                          * It's on the free list.  Remove and
971                          * arrange for it to be destroyed.
972                          */
973                         sn_rmfree(np);
974                         mutex_exit(&smbfreelist_lock);
975
976                         /*
977                          * Last part of sn_rmhash_locked().
978                          * NB: avl_destroy_nodes has already
979                          * removed this from the "hash" AVL.
980                          */
981                         mutex_enter(&np->r_statelock);
982                         np->r_flags &= ~RHASHED;
983                         mutex_exit(&np->r_statelock);
```



```
985                         /*
986                          * Add to the list of nodes to destroy.
987                          * Borrowing avl_child[0] for this list.
988                          */
989                         np->r_avl_node.avl_child[0] =
990                             (struct avl_node *)rlist;
991                         rlist = np;
992                 }
993         }
994         avl_destroy(&mi->smi_hash_avl);
995
996         /*
997          * Replace the (now destroyed) "hash" AVL with the
998          * temporary AVL, which restores the busy nodes.
999          */
1000         mi->smi_hash_avl = tmp_avl;
1001         rw_exit(&mi->smi_hash_lk);
1002
1003         /*
1004          * Now destroy the nodes on our temporary list (rlist).
1005          * This call to smbfs_addfree will end up destroying the
1006          * smbnode, but in a safe way with the appropriate set
1007          * of checks done.
1008          */
1009         while ((np = rlist) != NULL) {
1010                 rlist = (smbnode_t *)np->r_avl_node.avl_child[0];
1011                 smbfs_addfree(np);
1012         }
1013 }
1014
1015 /*
1016  * This routine destroys all the resources associated with the smbnode
1017  * and then the smbnode itself.  Note: sn_inactive has been called.
1018  *
1019  * NFS: nfs_subr.c:destroy_rnode
1020  */
1021 static void
1022 sn_destroy_node(smbnode_t *np)
1023 {
1024         vnode_t *vp;
1025         vfs_t *vfsp;
1026
1027         vp = SMBTOV(np);
1028         vfsp = vp->v_vfsp;
1029
1030         ASSERT(vp->v_count == 1);
1031         ASSERT(np->r_count == 0);
1032         ASSERT(np->r_mapcnt == 0);
1033         ASSERT(np->r_secattr.vsa_aclentp == NULL);
1034         ASSERT(np->r_cred == NULL);
1035         ASSERT(np->n_rpath == NULL);
1036         ASSERT(!(np->r_flags & RHASHED));
1037         ASSERT(np->r_freef == NULL && np->r_freeb == NULL);
1038         atomic_add_long((ulong_t *)&smbnodenew, -1);
1039         vn_invalid(vp);
1040         vn_free(vp);
1041         kmem_cache_free(smbnode_cache, np);
1042         VFS_RELE(vfsp);
1043 }
1044
1045 /*
1046  * Correspond to rflush() in NFS.
1047 #endif /* ! codereview */
1048  * Flush all vnodes in this (or every) vfs.
1049  * Used by smbfs_sync and by smbfs_unmount.
 180  * Used by nfs_sync and by nfs_unmount.
```

```
1050  */
1051 /*ARGSUSED*/
1052 void
1053 smbfs_rflush(struct vfs *vfsp, cred_t *cr) {

1055     smbmntinfo_t *mi;
1056     smbnode_t *np;
1057     vnode_t *vp;

1059     long num, cnt;

1061     vnode_t **vplist;

1063     if(vfsp == NULL)
1064         return;

1066     mi = VFTOSMI(vfsp);

1068     cnt = 0;

1070     num = mi->smi_hash_avl.avl_numnodes;

1072     vplist = kmem_alloc(num * sizeof (vnode_t*), KM_SLEEP);

1074     rw_enter(&mi->smi_hash_lk, RW_READER);
1075     for (np = avl_first(&mi->smi_hash_avl); np != NULL;
1076             np = avl_walk(&mi->smi_hash_avl, np, AVL_AFTER)) {
1077         vp = SMBTOV(np);
1078         if (vn_is_readonly(vp))
1079             continue;

1081         if (vn_has_cached_data(vp) && (np->r_flags & RDIRTY || np->r_mapcnt > 0)
1082             VN_HOLD(vp);
1083             vplist[cnt++] = vp;
1084             if (cnt == num)
1085                 break;
1086         }
1087     }
1088     rw_exit(&mi->smi_hash_lk);

1090     while (cnt-- > 0) {
1091         vp = vplist[cnt];
1092         (void) VOP_PUTPAGE(vp, 0, 0, 0, cr, NULL);
1093         VN_RELE(vp);
1094     }

1096     kmem_free(vplist, num * sizeof (vnode_t*));
 184 smbfs_rflush(struct vfs *vfsp, cred_t *cr)
 185 {
 186         /* Todo: mmap support. */
1097 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
  103466 Sat Aug 18 10:48:44 2012
new/usr/src/uts/common/fs/smbclnt/smbfs/smbfs_vnops.c
*** NO COMMENTS ***
**********************************************************
```

```
/*
 * Copyright (c) 2000-2001 Boris Popov
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    This product includes software developed by Boris Popov.
 * 4. Neither the name of the author nor the names of any co-contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ''AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * $Id: smbfs_vnops.c,v 1.128.36.1 2005/05/27 02:35:28 lindak Exp $
 */

/*
 * Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
 */

#include <sys/systm.h>
#include <sys/cred.h>
#include <sys/vnode.h>
#include <sys/vfs.h>
#include <sys/filio.h>
#include <sys/uio.h>
#include <sys/dirent.h>
#include <sys/errno.h>
#include <sys/sunddi.h>
#include <sys/sysmacros.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/vfs_opreg.h>
#include <sys/policy.h>

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



```
#include <vm/seg_map.h>
#include <vm/seg_kmem.h>
#include <vm/seg_kpm.h>

#endif /* ! codereview */
#include <netsmb/smb_osdep.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 <sys/fs/smbfs_ioctl.h>
#include <fs/fs_subr.h>

/*
 * We assign directory offsets like the NFS client, where the
 * offset increments by _one_ after each directory entry.
 * Further, the entries "." and ".." are always at offsets
 * zero and one (respectively) and the "real" entries from
 * the server appear at offsets starting with two.  This
 * macro is used to initialize the n_dirofs field after
 * setting n_dirseq with a _findopen call.
 */
#define FIRST_DIROFS    2

/*
 * These characters are illegal in NTFS file names.
 * ref: http://support.microsoft.com/kb/147438
 *
 * Careful!  The check in the XATTR case skips the
 * first character to allow colon in XATTR names.
 */
static const char illegal_chars[] = {
        ':',    /* colon - keep this first! */
        '\\',   /* back slash */
        '/',    /* slash */
        '*',    /* asterisk */
        '?',    /* question mark */
        '"',    /* double quote */
        '<',    /* less than sign */
        '>',    /* greater than sign */
        '|',    /* vertical bar */
        0
};

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

/* local static function defines */

static int      smbfslookup_cache(vnode_t *, char *, int, vnode_t **,
                        cred_t *);
static int      smbfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
                        int cache_ok, caller_context_t *);
static int      smbfsrename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm,
                        cred_t *cr, caller_context_t *);
static int      smbfssetattr(vnode_t *, struct vattr *, int, cred_t *);
static int      smbfs_accessx(void *, int, cred_t *);
static int      smbfs_readvdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
```

```
128                         caller_context_t *);
129 static void     smbfs_rele_fid(smbnode_t *, struct smb_cred *);

131 /*
132  * These are the vnode ops routines which implement the vnode interface to
133  * the networked file system.  These routines just take their parameters,
134  * make them look networkish by putting the right info into interface structs,
135  * and then calling the appropriate remote routine(s) to do the work.
136  *
137  * Note on directory name lookup cacheing:  If we detect a stale fhandle,
138  * we purge the directory cache relative to that vnode.  This way, the
139  * user won't get burned by the cache repeatedly.  See <smbfs/smbnode.h> for
140  * more details on smbnode locking.
141  */

143 static int      smbfs_open(vnode_t **, int, cred_t *, caller_context_t *);
144 static int      smbfs_close(vnode_t *, int, int, offset_t, cred_t *,
145                         caller_context_t *);
146 static int      smbfs_read(vnode_t *, struct uio *, int, cred_t *,
147                         caller_context_t *);
148 static int      smbfs_write(vnode_t *, struct uio *, int, cred_t *,
149                         caller_context_t *);
150 static int      smbfs_ioctl(vnode_t *, int, intptr_t, int, cred_t *, int *,
151                         caller_context_t *);
152 static int      smbfs_getattr(vnode_t *, struct vattr *, int, cred_t *,
153                         caller_context_t *);
154 static int      smbfs_setattr(vnode_t *, struct vattr *, int, cred_t *,
155                         caller_context_t *);
156 static int      smbfs_access(vnode_t *, int, int, cred_t *, caller_context_t *);
157 static int      smbfs_fsync(vnode_t *, int, cred_t *, caller_context_t *);
158 static void     smbfs_inactive(vnode_t *, cred_t *, caller_context_t *);
159 static int      smbfs_lookup(vnode_t *, char *, vnode_t **, struct pathname *,
160                         int, vnode_t *, cred_t *, caller_context_t *,
161                         int *, pathname_t *);
162 static int      smbfs_create(vnode_t *, char *, struct vattr *, enum vcexcl,
163                         int, vnode_t **, cred_t *, int, caller_context_t *,
164                         vsecattr_t *);
165 static int      smbfs_remove(vnode_t *, char *, cred_t *, caller_context_t *,
166                         int);
167 static int      smbfs_rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
168                         caller_context_t *, int);
169 static int      smbfs_mkdir(vnode_t *, char *, struct vattr *, vnode_t **,
170                         cred_t *, caller_context_t *, int, vsecattr_t *);
171 static int      smbfs_rmdir(vnode_t *, char *, vnode_t *, cred_t *,
172                         caller_context_t *, int);
173 static int      smbfs_readdir(vnode_t *, struct uio *, cred_t *, int *,
174                         caller_context_t *, int);
175 static int      smbfs_rwlock(vnode_t *, int, caller_context_t *);
176 static void     smbfs_rwunlock(vnode_t *, int, caller_context_t *);
177 static int      smbfs_seek(vnode_t *, offset_t, offset_t *, caller_context_t *);
178 static int      smbfs_frlock(vnode_t *, int, struct flock64 *, int, offset_t,
179                         struct flk_callback *, cred_t *, caller_context_t *);
180 static int      smbfs_space(vnode_t *, int, struct flock64 *, int, offset_t,
181                         cred_t *, caller_context_t *);
182 static int      smbfs_pathconf(vnode_t *, int, ulong_t *, cred_t *,
183                         caller_context_t *);
184 static int      smbfs_setsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
185                         caller_context_t *);
186 static int      smbfs_getsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
187                         caller_context_t *);
188 static int      smbfs_shrlock(vnode_t *, int, struct shrlock *, int, cred_t *,
189                         caller_context_t *);

191 static int uio_page_mapin(uio_t *uiop, page_t *pp);

193 static void uio_page_mapout(uio_t *uiop, page_t *pp);
```



```
195 static int smbfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
196         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
197         caller_context_t *ct);

199 static int smbfs_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
200         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
201         caller_context_t *ct);

203 static int smbfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
204         size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
205         caller_context_t *ct);

207 static int smbfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags,
208         cred_t *cr, caller_context_t *ct);

210 static int smbfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *len
211         int flags, cred_t *cr);

213 static int smbfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
214         page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
215         enum seg_rw rw, cred_t *cr, caller_context_t *ct);

217 static int smbfs_getapage(vnode_t *vp, u_offset_t off, size_t len,
218         uint_t *protp, page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
219         enum seg_rw rw, cred_t *cr);

221 static int writenp(smbnode_t *np, caddr_t base, int tcount, struct uio *uiop, in

223 #endif /* ! codereview */
224 /* Dummy function to use until correct function is ported in */
225 int noop_vnodeop() {
226         return (0);
227 }

229 struct vnodeops *smbfs_vnodeops = NULL;

231 /*
232  * Most unimplemented ops will return ENOSYS because of fs_nosys().
233  * The only ops where that won't work are ACCESS (due to open(2)
234  * failures) and ... (anything else left?)
235  */
236 const fs_operation_def_t smbfs_vnodeops_template[] = {
237         { VOPNAME_OPEN,         { .vop_open = smbfs_open } },
238         { VOPNAME_CLOSE,        { .vop_close = smbfs_close } },
239         { VOPNAME_READ,         { .vop_read = smbfs_read } },
240         { VOPNAME_WRITE,        { .vop_write = smbfs_write } },
241         { VOPNAME_IOCTL,        { .vop_ioctl = smbfs_ioctl } },
242         { VOPNAME_GETATTR,      { .vop_getattr = smbfs_getattr } },
243         { VOPNAME_SETATTR,      { .vop_setattr = smbfs_setattr } },
244         { VOPNAME_ACCESS,       { .vop_access = smbfs_access } },
245         { VOPNAME_LOOKUP,       { .vop_lookup = smbfs_lookup } },
246         { VOPNAME_CREATE,       { .vop_create = smbfs_create } },
247         { VOPNAME_REMOVE,       { .vop_remove = smbfs_remove } },
248         { VOPNAME_LINK,         { .error = fs_nosys } }, /* smbfs_link, */
249         { VOPNAME_RENAME,       { .vop_rename = smbfs_rename } },
250         { VOPNAME_MKDIR,        { .vop_mkdir = smbfs_mkdir } },
251         { VOPNAME_RMDIR,        { .vop_rmdir = smbfs_rmdir } },
252         { VOPNAME_READDIR,      { .vop_readdir = smbfs_readdir } },
253         { VOPNAME_SYMLINK,      { .error = fs_nosys } }, /* smbfs_symlink, */
254         { VOPNAME_READLINK,     { .error = fs_nosys } }, /* smbfs_readlink, */
255         { VOPNAME_FSYNC,        { .vop_fsync = smbfs_fsync } },
256         { VOPNAME_INACTIVE,     { .vop_inactive = smbfs_inactive } },
257         { VOPNAME_FID,          { .error = fs_nosys } }, /* smbfs_fid, */
258         { VOPNAME_RWLOCK,       { .vop_rwlock = smbfs_rwlock } },
259         { VOPNAME_RWUNLOCK,     { .vop_rwunlock = smbfs_rwunlock } },
```

```
260         { VOPNAME_SEEK,         { .vop_seek = smbfs_seek } },
261         { VOPNAME_FRLOCK,       { .vop_frlock = smbfs_frlock } },
262         { VOPNAME_SPACE,        { .vop_space = smbfs_space } },
263         { VOPNAME_REALVP,       { .error = fs_nosys } }, /* smbfs_realvp, */
264         { VOPNAME_GETPAGE,      { .vop_getpage = smbfs_getpage } }, /* smbfs_get
265         { VOPNAME_PUTPAGE,      { .vop_putpage = smbfs_putpage } }, /* smbfs_put
266         { VOPNAME_MAP,          { .vop_map = smbfs_map } }, /* smbfs_map, */
267         { VOPNAME_ADDMAP,       { .vop_addmap = smbfs_addmap } }, /* smbfs_addma
268         { VOPNAME_DELMAP,       { .vop_delmap = smbfs_delmap } }, /* smbfs_delma
269         { VOPNAME_DISPOSE,      { .vop_dispose = fs_dispose}},
 54         { VOPNAME_GETPAGE,      { .error = fs_nosys } }, /* smbfs_getpage, */
 55         { VOPNAME_PUTPAGE,      { .error = fs_nosys } }, /* smbfs_putpage, */
 56         { VOPNAME_MAP,          { .error = fs_nosys } }, /* smbfs_map, */
 57         { VOPNAME_ADDMAP,       { .error = fs_nosys } }, /* smbfs_addmap, */
 58         { VOPNAME_DELMAP,       { .error = fs_nosys } }, /* smbfs_delmap, */
270         { VOPNAME_DUMP,         { .error = fs_nosys } }, /* smbfs_dump, */
271         { VOPNAME_PATHCONF,     { .vop_pathconf = smbfs_pathconf } },
272         { VOPNAME_PAGEIO,       { .error = fs_nosys } }, /* smbfs_pageio, */
273         { VOPNAME_SETSECATTR,   { .vop_setsecattr = smbfs_setsecattr } },
274         { VOPNAME_GETSECATTR,   { .vop_getsecattr = smbfs_getsecattr } },
275         { VOPNAME_SHRLOCK,      { .vop_shrlock = smbfs_shrlock } },
276         { NULL, NULL }
277 };
```

unchanged_portion_omitted_

```
462 /*ARGSUSED*/
463 static int
464 smbfs_close(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
465         caller_context_t *ct)
466 {
467         smbnode_t       *np;
468         smbmntinfo_t    *smi;
469         struct smb_cred scred;

471         np = VTOSMB(vp);
472         smi = VTOSMI(vp);

474         /*
475          * Don't "bail out" for VFS_UNMOUNTED here,
476          * as we want to do cleanup, etc.
477          */

479         /*
480          * zone_enter(2) prevents processes from changing zones with SMBFS files
481          * open; if we happen to get here from the wrong zone we can't do
482          * anything over the wire.
483          */
484         if (smi->smi_zone_ref.zref_zone != curproc->p_zone) {
485                 /*
486                  * We could attempt to clean up locks, except we're sure
487                  * that the current process didn't acquire any locks on
488                  * the file: any attempt to lock a file belong to another zone
489                  * will fail, and one can't lock an SMBFS file and then change
490                  * zones, as that fails too.
491                  *
492                  * Returning an error here is the sane thing to do.  A
493                  * subsequent call to VN_RELE() which translates to a
494                  * smbfs_inactive() will clean up state: if the zone of the
495                  * vnode's origin is still alive and kicking, an async worker
496                  * thread will handle the request (from the correct zone), and
497                  * everything (minus the final smbfs_getattr_otw() call) should
498                  * be OK. If the zone is going away smbfs_async_inactive() will
499                  * throw away cached pages inline.
500                  */
501                 return (EIO);
502         }
```



```
504         /*
505          * If we are using local locking for this filesystem, then
506          * release all of the SYSV style record locks.  Otherwise,
507          * we are doing network locking and we need to release all
508          * of the network locks.  All of the locks held by this
509          * process on this file are released no matter what the
510          * incoming reference count is.
511          */
512         if (smi->smi_flags & SMI_LLOCK) {
513                 pid_t pid = ddi_get_pid();
514                 cleanlocks(vp, pid, 0);
515                 cleanshares(vp, pid);
516         }

518         /*
519          * This (passed in) count is the ref. count from the
520          * user's file_t before the closef call (fio.c).
521          * We only care when the reference goes away.
522          */
523         if (count > 1)
524                 return (0);

526         /*
527          * Decrement the reference count for the FID
528          * and possibly do the OtW close.
529          *
530          * Exclusive lock for modifying n_fid stuff.
531          * Don't want this one ever interruptible.
532          */
533         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
534         smb_credinit(&scred, cr);

536         /*
537          * If FID ref. count is 1 and count of mmaped pages isn't 0,
538          * we won't call smbfs_rele_fid(), because it will result in the otW clo
539          * The count of mapped pages isn't 0, which means the mapped pages
540          * possibly will be accessed after close(), we should keep the FID valid
541          * i.e., dont do the otW close.
542          * Dont worry that FID will be leaked, because when the
543          * vnode's count becomes 0, smbfs_inactive() will
544          * help us release FID and eventually do the otW close.
545          */
546         if (np->n_fidrefs > 1) {
547 #endif /* ! codereview */
548                 smbfs_rele_fid(np, &scred);
549         } else if (np->r_mapcnt == 0) {
550                 /*
551                  * Before otW close, make sure dirty pages written back.
552                  */
553                 if ((flag & FWRITE) && vn_has_cached_data(vp)) {
554                         /* smbfs_putapage() will acquire shared lock, so release
555                          * exclusive lock temporally.
556                          */
557                         smbfs_rw_exit(&np->r_lkserlock);

559                         (void) smbfs_putpage(vp, (offset_t) 0, 0, B_INVAL | B_AS

561                         /* acquire exclusive lock again. */
562                         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0
563                 }
564                 smbfs_rele_fid(np, &scred);
565         }
566 #endif /* ! codereview */

568         smb_credrele(&scred);
```

```
569         smbfs_rw_exit(&np->r_lkserlock);
570
571         return (0);
572 }
573
574 /*
575  * Helper for smbfs_close.  Decrement the reference count
576  * for an SMB-level file or directory ID, and when the last
577  * reference for the fid goes away, do the OtW close.
578  * Also called in smbfs_inactive (defensive cleanup).
579  */
580 static void
581 smbfs_rele_fid(smbnode_t *np, struct smb_cred *scred)
582 {
583         smb_share_t     *ssp;
584         cred_t          *oldcr;
585         struct smbfs_fctx *fctx;
586         int             error;
587         uint16_t ofid;
588
589         ssp = np->n_mount->smi_share;
590         error = 0;
591
592         /* Make sure we serialize for n_dirseq use. */
593         ASSERT(smbfs_rw_lock_held(&np->r_lkserlock, RW_WRITER));
594
595         /*
596          * Note that vp->v_type may change if a remote node
597          * is deleted and recreated as a different type, and
598          * our getattr may change v_type accordingly.
599          * Now use n_ovtype to keep track of the v_type
600          * we had during open (see comments above).
601          */
602         switch (np->n_ovtype) {
603         case VDIR:
604                 ASSERT(np->n_dirrefs > 0);
605                 if (--np->n_dirrefs)
606                         return;
607                 if ((fctx = np->n_dirseq) != NULL) {
608                         np->n_dirseq = NULL;
609                         np->n_dirofs = 0;
610                         error = smbfs_smb_findclose(fctx, scred);
611                 }
612                 break;
613
614         case VREG:
615                 ASSERT(np->n_fidrefs > 0);
616                 if (--np->n_fidrefs)
617                         return;
618                 if ((ofid = np->n_fid) != SMB_FID_UNUSED) {
619                         np->n_fid = SMB_FID_UNUSED;
620                         /* After reconnect, n_fid is invalid */
621                         if (np->n_vcgenid == ssp->ss_vcgenid) {
622                                 error = smbfs_smb_close(
623                                     ssp, ofid, NULL, scred);
624                         }
625                 }
626                 break;
627
628         default:
629                 SMBVDEBUG("bad n_ovtype %d\n", np->n_ovtype);
630                 break;
631         }
632         if (error) {
633                 SMBVDEBUG("error %d closing %s\n",
634                     error, np->n_rpath);
```



```
635         }
636
637         /* Allow next open to use any v_type. */
638         np->n_ovtype = VNON;
639
640         /*
641          * Other "last close" stuff.
642          */
643         mutex_enter(&np->r_statelock);
644         if (np->n_flag & NATTRCHANGED)
645                 smbfs_attrcache_rm_locked(np);
646         oldcr = np->r_cred;
647         np->r_cred = NULL;
648         mutex_exit(&np->r_statelock);
649         if (oldcr != NULL)
650                 crfree(oldcr);
651 }
652
653 /* ARGSUSED */
654 static int
655 smbfs_read(vnode_t * vp, struct uio * uiop, int ioflag, cred_t * cr,
656             caller_context_t * ct)
325 smbfs_read(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
326         caller_context_t *ct)
657 {
658         struct smb_cred scred;
659         struct vattr    va;
660         smbnode_t      *np;
661         smbmntinfo_t   *smi;
662         smb_share_t    *ssp;
663         offset_t        endoff;
664         ssize_t         past_eof;
665         int             error;
666
667         caddr_t         base;
668         u_offset_t      blk;
669         u_offset_t      boff;
670         size_t          blen;
671         uint_t          flags;
672
673 #endif /* ! codereview */
674         np = VTOSMB(vp);
675         smi = VTOSMI(vp);
676         ssp = smi->smi_share;
677
678         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
679                 return (EIO);
680
681         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
682                 return (EIO);
683
684         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_READER));
685
686         if (vp->v_type != VREG)
687                 return (EISDIR);
688
689         if (uiop->uio_resid == 0)
690                 return (0);
691
692         /*
693          * Like NFS3, just check for 63-bit overflow.  Our SMB layer takes
694          * care to return EFBIG when it has to fallback to a 32-bit call.
337          * Like NFS3, just check for 63-bit overflow.
338          * Our SMB layer takes care to return EFBIG
339          * when it has to fallback to a 32-bit call.
695          */
```

```
696         endoff = uiop->uio_loffset + uiop->uio_resid;
697         if (uiop->uio_loffset < 0 || endoff < 0)
698                 return (EINVAL);

700         /* get vnode attributes from server */
701         va.va_mask = AT_SIZE | AT_MTIME;
702         if (error = smbfsgetattr(vp, &va, cr))
703                 return (error);

705         /* Update mtime with mtime from server here? */

707         /* if offset is beyond EOF, read nothing */
708         if (uiop->uio_loffset >= va.va_size)
709                 return (0);

711         /*
712          * Limit the read to the remaining file size. Do this by temporarily
713          * reducing uio_resid by the amount the lies beyoned the EOF.
357          * Limit the read to the remaining file size.
358          * Do this by temporarily reducing uio_resid
359          * by the amount the lies beyoned the EOF.
714          */
715         if (endoff > va.va_size) {
716                 past_eof = (ssize_t) (endoff - va.va_size);
362                 past_eof = (ssize_t)(endoff - va.va_size);
717                 uiop->uio_resid -= past_eof;
718         } else
719                 past_eof = 0;

721         /* Bypass the VM if vnode is non-cacheable. */
722         if ((vp->v_flag & VNOCACHE) ||
723             ((np->r_flags & RDIRECTIO) &&
724              np->r_mapcnt == 0 &&
725              !(vn_has_cached_data(vp)))) {

727 #endif /* ! codereview */
728                 /* Shared lock for n_fid use in smb_rwuio */
729                 if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp))
730                         return (EINTR);
731                 smb_credinit(&scred, cr);

733                 /* After reconnect, n_fid is invalid */
734                 if (np->n_vcgenid != ssp->ss_vcgenid)
735                         error = ESTALE;
736                 else
737                         error = smb_rwuio(ssp, np->n_fid, UIO_READ,
738                                           uiop, &scred, smb_timo_read);

740                 smb_credrele(&scred);
741                 smbfs_rw_exit(&np->r_lkserlock);

743         } else {

745                 /* Do I/O through segmap. */
746                 do {
747                         blk = uiop->uio_loffset & MAXBMASK;
748                         boff = uiop->uio_loffset & MAXBOFFSET;
749                         blen = MIN(MAXBSIZE - boff, uiop->uio_resid);

751                         if (vpm_enable) {

753                                 error = vpm_data_copy(vp, blk + boff, blen, uiop

755                         } else {

757                                 base = segmap_getmapflt(segkmap, vp, blk + boff,
```



```
759                                 error = uiomove(base + boff, blen, UIO_READ, uio
760                         }

762                         if (!error) {
763                                 mutex_enter(&np->r_statelock);
764                                 if ((blen + boff == MAXBSIZE) || (uiop->uio_loff
765                                         flags = SM_DONTNEED;
766                                 } else {
767                                         flags = 0;
768                                 }
769                                 mutex_exit(&np->r_statelock);
770                         } else {
771                                 flags = 0;
772                         }
773                         if (vpm_enable) {
774                                 (void) vpm_sync_pages(vp, blk + boff, blen, flag
775                         } else {
776                                 (void) segmap_release(segkmap, base, flags);
777                         }
778                 } while (!error && uiop->uio_resid > 0);
779         }

781 #endif /* ! codereview */
782         /* undo adjustment of resid */
783         uiop->uio_resid += past_eof;

785         return (error);
786 }


788 /* ARGSUSED */
789 static int
790 smbfs_write(vnode_t * vp, struct uio * uiop, int ioflag, cred_t * cr,
791             caller_context_t * ct)
370 smbfs_write(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
371         caller_context_t *ct)
792 {
793         struct smb_cred scred;
794         struct vattr    va;
795         smbnode_t      *np;
796         smbmntinfo_t   *smi;
797         smb_share_t    *ssp;
798         offset_t        endoff, limit;
799         ssize_t         past_limit;
800         int             error, timo;

802         caddr_t         base;
803         u_offset_t      blk;
804         u_offset_t      boff;
805         size_t          blen;
806         uint_t          flags;

808         u_offset_t      last_off;
809         size_t          last_resid;

811 #endif /* ! codereview */
812         np = VTOSMB(vp);
813         smi = VTOSMI(vp);
814         ssp = smi->smi_share;

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

819         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
820                 return (EIO);
```

```
822         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_WRITER));
823
824         if (vp->v_type != VREG)
825                 return (EISDIR);
826
827         if (uiop->uio_resid == 0)
828                 return (0);
829
830         /*
831          * Handle ioflag bits: (FAPPEND|FSYNC|FDSYNC)
832          */
833         if (ioflag & (FAPPEND | FSYNC)) {
834                 if (np->n_flag & NMODIFIED) {
835                         smbfs_attrcache_remove(np);
836                         /* XXX: smbfs_vinvalbuf? */
837                 }
838         }
839         if (ioflag & FAPPEND) {
840                 /*
841                  * File size can be changed by another client
842                  */
843                 va.va_mask = AT_SIZE;
844                 if (error = smbfsgetattr(vp, &va, cr))
845                         return (error);
846                 uiop->uio_loffset = va.va_size;
847         }
848         /*
849          * Like NFS3, just check for 63-bit overflow.
850          */
851         endoff = uiop->uio_loffset + uiop->uio_resid;
852         if (uiop->uio_loffset < 0 || endoff < 0)
853                 return (EINVAL);
854
855         /*
856          * Check to make sure that the process will not exceed its limit on
857          * file size.  It is okay to write up to the limit, but not beyond.
858          * Thus, the write which reaches the limit will be short and the next
859          * write will return an error.
860          *
861          * So if we're starting at or beyond the limit, EFBIG. Otherwise,
862          * temporarily reduce resid to the amount the falls after the limit.
391          * Check to make sure that the process will not exceed
392          * its limit on file size.  It is okay to write up to
393          * the limit, but not beyond.  Thus, the write which
394          * reaches the limit will be short and the next write
395          * will return an error.
396          *
397          * So if we're starting at or beyond the limit, EFBIG.
398          * Otherwise, temporarily reduce resid to the amount
399          * the falls after the limit.
863          */
864         limit = uiop->uio_llimit;
865         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
866                 limit = MAXOFFSET_T;
867         if (uiop->uio_loffset >= limit)
868                 return (EFBIG);
869         if (endoff > limit) {
870                 past_limit = (ssize_t) (endoff - limit);
407                 past_limit = (ssize_t)(endoff - limit);
871                 uiop->uio_resid -= past_limit;
872         } else
873                 past_limit = 0;
874
875         /* Bypass the VM if vnode is non-cacheable. */
```



```
876         if ((vp->v_flag & VNOCACHE) ||
877             ((np->r_flags & RDIRECTIO) &&
878              np->r_mapcnt == 0 &&
879              !(vn_has_cached_data(vp)))) {
880
881 #endif /* ! codereview */
882                 /* Timeout: longer for append. */
883                 timo = smb_timo_write;
884                 if (endoff > np->r_size)
885                         timo = smb_timo_append;
886
887                 /* Shared lock for n_fid use in smb_rwuio */
888                 if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp))
889                         return (EINTR);
890                 smb_credinit(&scred, cr);
891
892                 /* After reconnect, n_fid is invalid */
893                 if (np->n_vcgenid != ssp->ss_vcgenid)
894                         error = ESTALE;
895                 else
896                         error = smb_rwuio(ssp, np->n_fid, UIO_WRITE,
897                                           uiop, &scred, timo);
898
899                 if (error == 0) {
900                         mutex_enter(&np->r_statelock);
901                         np->n_flag |= (NFLUSHWIRE | NATTRCHANGED);
902                         if (uiop->uio_loffset > (offset_t) np->r_size)
903                                 np->r_size = (len_t) uiop->uio_loffset;
412                 if (uiop->uio_loffset > (offset_t)np->r_size)
413                         np->r_size = (len_t)uiop->uio_loffset;
904                         mutex_exit(&np->r_statelock);
905                         if (ioflag & (FSYNC | FDSYNC)) {
415                 if (ioflag & (FSYNC|FDSYNC)) {
906                                 /* Don't error the I/O if this fails. */
907                                 (void) smbfs_smb_flush(np, &scred);
908                         }
909                 }
910                 smb_credrele(&scred);
911                 smbfs_rw_exit(&np->r_lkserlock);
912
913         } else {
914
915                 /* Do I/O through segmap. */
916                 size_t          bsize = vp->v_vfsp->vfs_bsize;
917
918                 do {
919                         blk = uiop->uio_loffset & MAXBMASK;
920                         boff = uiop->uio_loffset & MAXBOFFSET;
921                         blen = MIN(MAXBSIZE - boff, uiop->uio_resid);
922
923                         last_off = uiop->uio_loffset;
924                         last_resid = uiop->uio_resid;
925
926                         uio_prefaultpages((ssize_t) blen, uiop);
927
928                         if (vpm_enable) {
929
930                                 error = writenp(np, NULL, blen, uiop, 0);
931
932                         } else {
933
934                                 if (segmap_kpm) {
935                                         u_offset_t      poff = uiop->uio_loffset
936                                         size_t          plen = MIN(PAGESIZE - po
```

```
 938                                        int             pagecreate;
 940                                        mutex_enter(&np->r_statelock);
 941                                        pagecreate = (poff == 0) &&
 942                                                ((plen == PAGESIZE) ||
 943                                                 (uiop->uio_loffset + plen >= np
 944                                        mutex_exit(&np->r_statelock);
 946                                        base = segmap_getmapflt(segkmap, vp, blk
 947                                        error = writenp(np, base + poff, blen, u
 949                                } else {
 950                                        base = segmap_getmapflt(segkmap, vp, blk
 951                                        error = writenp(np, base + boff, blen, u
 952                                }
 953                        }
 955                        if (!error) {
 956                                if (uiop->uio_loffset % bsize == 0) {
 957                                        flags = SM_WRITE | SM_DONTNEED;
 958                                } else {
 959                                        flags = 0;
 960                                }
 962                                if (ioflag & (FSYNC | FDSYNC)) {
 963                                        flags &= ~SM_ASYNC;
 964                                        flags |= SM_WRITE;
 965                                }
 966                                if (vpm_enable) {
 967                                        error = vpm_sync_pages(vp, blk, blen, fl
 968                                } else {
 969                                        error = segmap_release(segkmap, base, fl
 970                                }
 971                        } else {
 972                                if (vpm_enable) {
 973                                        (void) vpm_sync_pages(vp, blk, blen, 0);
 974                                } else {
 975                                        (void) segmap_release(segkmap, base, 0);
 976                                }
 977                        }
 978                } while (!error && uiop->uio_resid > 0);
 979        }
 981 #endif /* ! codereview */
 982        /* undo adjustment of resid */
 983        if (error) {
 984                uiop->uio_resid = last_resid + past_limit;
 985                uiop->uio_loffset = last_off;
 986        } else {
 987 #endif /* ! codereview */
 988                uiop->uio_resid += past_limit;
 989        }
 990 #endif /* ! codereview */
 992        return (error);
 993 }
 995 /* correspond to writerp() in nfs_client.c */
 996 static int
 997 writenp(smbnode_t * np, caddr_t base, int tcount, struct uio * uiop, int pgcreat
 998 {
 999        int             pagecreate;
1000        int             n;
1001        int             saved_n;
1002        caddr_t         saved_base;
1003        u_offset_t      offset;
```



```
1004        int             error;
1005        int             sm_error;
1007        vnode_t        *vp = SMBTOV(np);
1009        ASSERT(tcount <= MAXBSIZE && tcount <= uiop->uio_resid);
1010        ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_WRITER));
1011        if (!vpm_enable) {
1012                ASSERT(((uintptr_t) base & MAXBOFFSET) + tcount <= MAXBSIZE);
1013        }
1014        /*
1015         * Move bytes in at most PAGESIZE chunks. We must avoid spanning
1016         * pages in uiomove() because page faults may cause the cache to be
1017         * invalidated out from under us. The r_size is not updated until
1018         * after the uiomove. If we push the last page of a file before
1019         * r_size is correct, we will lose the data written past the current
1020         * (and invalid) r_size.
1021         */
1022        do {
1023                offset = uiop->uio_loffset;
1024                pagecreate = 0;
1026                /*
1027                 * n is the number of bytes required to satisfy the request
1028                 * or the number of bytes to fill out the page.
1029                 */
1030                n = (int) MIN((PAGESIZE - (offset & PAGEOFFSET)), tcount);
1032                /*
1033                 * Check to see if we can skip reading in the page and just
1034                 * allocate the memory. We can do this if we are going to
1035                 * rewrite the entire mapping or if we are going to write to
1036                 * or beyond the current end of file from the beginning of
1037                 * the mapping.
1038                 *
1039                 * The read of r_size is now protected by r_statelock.
1040                 */
1041                mutex_enter(&np->r_statelock);
1042                /*
1043                 * When pgcreated is nonzero the caller has already done a
1044                 * segmap_getmapflt with forcefault 0 and S_WRITE. With
1045                 * segkpm this means we already have at least one page
1046                 * created and mapped at base.
1047                 */
1048                pagecreate = pgcreated ||
1049                        ((offset & PAGEOFFSET) == 0 &&
1050                         (n == PAGESIZE || ((offset + n) >= np->r_size)));
1052                mutex_exit(&np->r_statelock);
1054                if (!vpm_enable && pagecreate) {
1055                        /*
1056                         * The last argument tells segmap_pagecreate() to
1057                         * always lock the page, as opposed to sometimes
1058                         * returning with the page locked. This way we avoid
1059                         * a fault on the ensuing uiomove(), but also more
1060                         * importantly (to fix bug 1094402) we can call
1061                         * segmap_fault() to unlock the page in all cases. An
1062                         * alternative would be to modify segmap_pagecreate()
1063                         * to tell us when it is locking a page, but that's a
1064                         * fairly major interface change.
1065                         */
1066                        if (pgcreated == 0)
1067                                (void) segmap_pagecreate(segkmap, base,
1068                                                         (uint_t) n, 1);
1069                        saved_base = base;
```

```
                        saved_n = n;
                }
                /*
                 * The number of bytes of data in the last page can not be
                 * accurately be determined while page is being uiomove'd to
                 * and the size of the file being updated. Thus, inform
                 * threads which need to know accurately how much data is in
                 * the last page of the file. They will not do the i/o
                 * immediately, but will arrange for the i/o to happen later
                 * when this modify operation will have finished.
                 */
                ASSERT(!(np->r_flags & RMODINPROGRESS));
                mutex_enter(&np->r_statelock);
                np->r_flags |= RMODINPROGRESS;
                np->r_modaddr = (offset & MAXBMASK);
                mutex_exit(&np->r_statelock);

                if (vpm_enable) {
                        /*
                         * Copy data. If new pages are created, part of the
                         * page that is not written will be initizliazed with
                         * zeros.
                         */
                        error = vpm_data_copy(vp, offset, n, uiop,
                                              !pagecreate, NULL, 0, S_WRITE);
                } else {
                        error = uiomove(base, n, UIO_WRITE, uiop);
                }

                /*
                 * r_size is the maximum number of bytes known to be in the
                 * file. Make sure it is at least as high as the first
                 * unwritten byte pointed to by uio_loffset.
                 */
                mutex_enter(&np->r_statelock);
                if (np->r_size < uiop->uio_loffset)
                        np->r_size = uiop->uio_loffset;
                np->r_flags &= ~RMODINPROGRESS;
                np->r_flags |= RDIRTY;
                mutex_exit(&np->r_statelock);

                /* n = # of bytes written */
                n = (int) (uiop->uio_loffset - offset);

                if (!vpm_enable) {
                        base += n;
                }
                tcount -= n;
                /*
                 * If we created pages w/o initializing them completely, we
                 * need to zero the part that wasn't set up. This happens on
                 * a most EOF write cases and if we had some sort of error
                 * during the uiomove.
                 */
                if (!vpm_enable && pagecreate) {
                        if ((uiop->uio_loffset & PAGEOFFSET) || n == 0)
                                (void) kzero(base, PAGESIZE - n);

                        if (pgcreated) {
                                /*
                                 * Caller is responsible for this page, it
                                 * was not created in this loop.
                                 */
                                pgcreated = 0;
                        } else {
                                /*
```



```
                                 * For bug 1094402: segmap_pagecreate locks
                                 * page. Unlock it. This also unlocks the
                                 * pages allocated by page_create_va() in
                                 * segmap_pagecreate().
                                 */
                                sm_error = segmap_fault(kas.a_hat, segkmap,
                                                        saved_base, saved_n,
                                                      F_SOFTUNLOCK, S_WRITE);
                                if (error == 0)
                                        error = sm_error;
                        }
                }
        } while (tcount > 0 && error == 0);

        return (error);
}
#endif /* ! codereview */

/* ARGSUSED */
static int
smbfs_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag,
        cred_t *cr, int *rvalp, caller_context_t *ct)
{
        int             error;
        smbmntinfo_t    *smi;

        smi = VTOSMI(vp);

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

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

        switch (cmd) {
                /* First three from ZFS. XXX - need these? */

        case _FIOFFS:
                error = smbfs_fsync(vp, 0, cr, ct);
                break;

                /*
                 * The following two ioctls are used by bfu.
                 * Silently ignore to avoid bfu errors.
                 */
        case _FIOGDIO:
        case _FIOSDIO:
                error = 0;
                break;

#ifdef NOT_YET  /* XXX - from the NFS code. */
        case _FIODIRECTIO:
                error = smbfs_directio(vp, (int)arg, cr);
#endif

                /*
                 * Allow get/set with "raw" security descriptor (SD) data.
                 * Useful for testing, diagnosing idmap problems, etc.
                 */
        case SMBFSIO_GETSD:
                error = smbfs_acl_iocget(vp, arg, flag, cr);
                break;

        case SMBFSIO_SETSD:
                error = smbfs_acl_iocset(vp, arg, flag, cr);
                break;
```

```
1203         default:
1204                 error = ENOTTY;
1205                 break;
1206         }

1208         return (error);
1209 }


1212 /*
1213  * Return either cached or remote attributes. If get remote attr
1214  * use them to check and invalidate caches, then cache the new attributes.
1215  *
1216  * XXX
1217  * This op should eventually support PSARC 2007/315, Extensible Attribute
1218  * Interfaces, for richer metadata.
1219  */
1220 /* ARGSUSED */
1221 static int
1222 smbfs_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1223         caller_context_t *ct)
1224 {
1225         smbnode_t *np;
1226         smbmntinfo_t *smi;

1228         smi = VTOSMI(vp);

1230         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1231                 return (EIO);

1233         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1234                 return (EIO);

1236         /*
1237          * If it has been specified that the return value will
1238          * just be used as a hint, and we are only being asked
1239          * for size, fsid or rdevid, then return the client's
1240          * notion of these values without checking to make sure
1241          * that the attribute cache is up to date.
1242          * The whole point is to avoid an over the wire GETATTR
1243          * call.
1244          */
1245         np = VTOSMB(vp);
1246         if (flags & ATTR_HINT) {
1247                 if (vap->va_mask ==
1248                     (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
1249                         mutex_enter(&np->r_statelock);
1250                         if (vap->va_mask | AT_SIZE)
1251                                 vap->va_size = np->r_size;
1252                         if (vap->va_mask | AT_FSID)
1253                                 vap->va_fsid = vp->v_vfsp->vfs_dev;
1254                         if (vap->va_mask | AT_RDEV)
1255                                 vap->va_rdev = vp->v_rdev;
1256                         mutex_exit(&np->r_statelock);
1257                         return (0);
1258                 }
1259         }

1261         return (smbfsgetattr(vp, vap, cr));
1262 }

1264 /* smbfsgetattr() in smbfs_client.c */

1266 /*
1267  * XXX
```



```
1268  * This op should eventually support PSARC 2007/315, Extensible Attribute
1269  * Interfaces, for richer metadata.
1270  */
1271 /*ARGSUSED4*/
1272 static int
1273 smbfs_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1274                 caller_context_t *ct)
1275 {
1276         vfs_t           *vfsp;
1277         smbmntinfo_t    *smi;
1278         int             error;
1279         uint_t          mask;
1280         struct vattr    oldva;

1282         vfsp = vp->v_vfsp;
1283         smi = VFTOSMI(vfsp);

1285         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1286                 return (EIO);

1288         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
1289                 return (EIO);

1291         mask = vap->va_mask;
1292         if (mask & AT_NOSET)
1293                 return (EINVAL);

1295         if (vfsp->vfs_flag & VFS_RDONLY)
1296                 return (EROFS);

1298         /*
1299          * This is a _local_ access check so that only the owner of
1300          * this mount can set attributes.  With ACLs enabled, the
1301          * file owner can be different from the mount owner, and we
1302          * need to check the _mount_ owner here.  See _access_rwx
1303          */
1304         bzero(&oldva, sizeof (oldva));
1305         oldva.va_mask = AT_TYPE | AT_MODE;
1306         error = smbfsgetattr(vp, &oldva, cr);
1307         if (error)
1308                 return (error);
1309         oldva.va_mask |= AT_UID | AT_GID;
1310         oldva.va_uid = smi->smi_uid;
1311         oldva.va_gid = smi->smi_gid;

1313         error = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
1314             smbfs_accessx, vp);
1315         if (error)
1316                 return (error);

1318         if (mask & (AT_UID | AT_GID)) {
1319                 if (smi->smi_flags & SMI_ACL)
1320                         error = smbfs_acl_setids(vp, vap, cr);
1321                 else
1322                         error = ENOSYS;
1323                 if (error != 0) {
1324                         SMBVDEBUG("error %d seting UID/GID on %s",
1325                             error, VTOSMB(vp)->n_rpath);
1326                         /*
1327                          * It might be more correct to return the
1328                          * error here, but that causes complaints
1329                          * when root extracts a cpio archive, etc.
1330                          * So ignore this error, and go ahead with
1331                          * the rest of the setattr work.
1332                          */
1333                 }
```

```
1334         }

1336         return (smbfssetattr(vp, vap, flags, cr));
1337 }

1339 /*
1340  * Mostly from Darwin smbfs_setattr()
1341  * but then modified a lot.
1342  */
1343 /* ARGSUSED */
1344 static int
1345 smbfssetattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr)
1346 {
1347         int             error = 0;
1348         smbnode_t       *np = VTOSMB(vp);
1349         uint_t          mask = vap->va_mask;
1350         struct timespec *mtime, *atime;
1351         struct smb_cred scred;
1352         int             cerror, modified = 0;
1353         unsigned short  fid;
1354         int have_fid = 0;
1355         uint32_t rights = 0;

1357         ASSERT(curproc->p_zone == VTOSMI(vp)->smi_zone_ref.zref_zone);

1359         /*
1360          * There are no settable attributes on the XATTR dir,
1361          * so just silently ignore these.  On XATTR files,
1362          * you can set the size but nothing else.
1363          */
1364         if (vp->v_flag & V_XATTRDIR)
1365                 return (0);
1366         if (np->n_flag & N_XATTR) {
1367                 if (mask & AT_TIMES)
1368                         SMBVDEBUG("ignore set time on xattr\n");
1369                 mask &= AT_SIZE;
1370         }

1372         /*
1373          * If our caller is trying to set multiple attributes, they
1374          * can make no assumption about what order they are done in.
1375          * Here we try to do them in order of decreasing likelihood
1376          * of failure, just to minimize the chance we'll wind up
1377          * with a partially complete request.
1378          */

1380         /* Shared lock for (possible) n_fid use. */
1381         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
1382                 return (EINTR);
1383         smb_credinit(&scred, cr);

1385         /*
1386          * Will we need an open handle for this setattr?
1387          * If so, what rights will we need?
1388          */
1389         if (mask & (AT_ATIME | AT_MTIME)) {
1390                 rights |=
1391                     SA_RIGHT_FILE_WRITE_ATTRIBUTES;
1392         }
1393         if (mask & AT_SIZE) {
1394                 rights |=
1395                     SA_RIGHT_FILE_WRITE_DATA |
1396                     SA_RIGHT_FILE_APPEND_DATA;
1397         }

1399         /*
```



```
1400          * Only SIZE really requires a handle, but it's
1401          * simpler and more reliable to set via a handle.
1402          * Some servers like NT4 won't set times by path.
1403          * Also, we're usually setting everything anyway.
1404          */
1405         if (mask & (AT_SIZE | AT_ATIME | AT_MTIME)) {
1406                 error = smbfs_smb_tmpopen(np, rights, &scred, &fid);
1407                 if (error) {
1408                         SMBVDEBUG("error %d opening %s\n",
1409                             error, np->n_rpath);
1410                         goto out;
1411                 }
1412                 have_fid = 1;
1413         }

1415         /*
1416          * If the server supports the UNIX extensions, right here is where
1417          * we'd support changes to uid, gid, mode, and possibly va_flags.
1418          * For now we claim to have made any such changes.
1419          */

1421         if (mask & AT_SIZE) {
1422                 /*
1423                  * If the new file size is less than what the client sees as
1424                  * the file size, then just change the size and invalidate
1425                  * the pages.
1426                  * I am commenting this code at present because the function
1427                  * smbfs_putapage() is not yet implemented.
1428                  */

1430                 /*
1431                  * Set the file size to vap->va_size.
1432                  */
1433                 ASSERT(have_fid);
1434                 error = smbfs_smb_setfsize(np, fid, vap->va_size, &scred);
1435                 if (error) {
1436                         SMBVDEBUG("setsize error %d file %s\n",
1437                             error, np->n_rpath);
1438                 } else {
1439                         /*
1440                          * Darwin had code here to zero-extend.
1441                          * Tests indicate the server will zero-fill,
1442                          * so looks like we don't need to do this.
1443                          * Good thing, as this could take forever.
1444                          *
1445                          * XXX: Reportedly, writing one byte of zero
1446                          * at the end offset avoids problems here.
1447                          */
1448                         mutex_enter(&np->r_statelock);
1449                         np->r_size = vap->va_size;
1450                         mutex_exit(&np->r_statelock);
1451                         modified = 1;
1452                 }
1453         }

1455         /*
1456          * XXX: When Solaris has create_time, set that too.
1457          * Note: create_time is different from ctime.
1458          */
1459         mtime = ((mask & AT_MTIME) ? &vap->va_mtime : 0);
1460         atime = ((mask & AT_ATIME) ? &vap->va_atime : 0);

1462         if (mtime || atime) {
1463                 /*
1464                  * Always use the handle-based set attr call now.
1465                  * Not trying to set DOS attributes here so pass zero.
```

```
1466                  */
1467                 ASSERT(have_fid);
1468                 error = smbfs_smb_setfattr(np, fid,
1469                     0, mtime, atime, &scred);
1470                 if (error) {
1471                         SMBVDEBUG("set times error %d file %s\n",
1472                             error, np->n_rpath);
1473                 } else {
1474                         modified = 1;
1475                 }
1476         }

1478 out:
1479         if (modified) {
1480                 /*
1481                  * Invalidate attribute cache in case the server
1482                  * doesn't set exactly the attributes we asked.
1483                  */
1484                 smbfs_attrcache_remove(np);
1485         }

1487         if (have_fid) {
1488                 cerror = smbfs_smb_tmpclose(np, fid, &scred);
1489                 if (cerror)
1490                         SMBVDEBUG("error %d closing %s\n",
1491                             cerror, np->n_rpath);
1492         }

1494         smb_credrele(&scred);
1495         smbfs_rw_exit(&np->r_lkserlock);

1497         return (error);
1498 }

1500 /*
1501  * smbfs_access_rwx()
1502  * Common function for smbfs_access, etc.
1503  *
1504  * The security model implemented by the FS is unusual
1505  * due to the current "single user mounts" restriction:
1506  * All access under a given mount point uses the CIFS
1507  * credentials established by the owner of the mount.
1508  *
1509  * Most access checking is handled by the CIFS server,
1510  * but we need sufficient Unix access checks here to
1511  * prevent other local Unix users from having access
1512  * to objects under this mount that the uid/gid/mode
1513  * settings in the mount would not allow.
1514  *
1515  * With this model, there is a case where we need the
1516  * ability to do an access check before we have the
1517  * vnode for an object.  This function takes advantage
1518  * of the fact that the uid/gid/mode is per mount, and
1519  * avoids the need for a vnode.
1520  *
1521  * We still (sort of) need a vnode when we call
1522  * secpolicy_vnode_access, but that only uses
1523  * the vtype field, so we can use a pair of fake
1524  * vnodes that have only v_type filled in.
1525  *
1526  * XXX: Later, add a new secpolicy_vtype_access()
1527  * that takes the vtype instead of a vnode, and
1528  * get rid of the tmpl_vxxx fake vnodes below.
1529  */
1530 static int
1531 smbfs_access_rwx(vfs_t *vfsp, int vtype, int mode, cred_t *cr)
```



```
1532 {
1533         /* See the secpolicy call below. */
1534         static const vnode_t tmpl_vdir = { .v_type = VDIR };
1535         static const vnode_t tmpl_vreg = { .v_type = VREG };
1536         vattr_t         va;
1537         vnode_t         *tvp;
1538         struct smbmntinfo *smi = VFTOSMI(vfsp);
1539         int shift = 0;

1541         /*
1542          * Build our (fabricated) vnode attributes.
1543          * XXX: Could make these templates in the
1544          * per-mount struct and use them here.
1545          */
1546         bzero(&va, sizeof (va));
1547         va.va_mask = AT_TYPE | AT_MODE | AT_UID | AT_GID;
1548         va.va_type = vtype;
1549         va.va_mode = (vtype == VDIR) ?
1550             smi->smi_dmode : smi->smi_fmode;
1551         va.va_uid = smi->smi_uid;
1552         va.va_gid = smi->smi_gid;

1554         /*
1555          * Disallow write attempts on read-only file systems,
1556          * unless the file is a device or fifo node.  Note:
1557          * Inline vn_is_readonly and IS_DEVVP here because
1558          * we may not have a vnode ptr.  Original expr. was:
1559          * (mode & VWRITE) && vn_is_readonly(vp) && !IS_DEVVP(vp))
1560          */
1561         if ((mode & VWRITE) &&
1562             (vfsp->vfs_flag & VFS_RDONLY) &&
1563             !(vtype == VCHR || vtype == VBLK || vtype == VFIFO))
1564                 return (EROFS);

1566         /*
1567          * Disallow attempts to access mandatory lock files.
1568          * Similarly, expand MANDLOCK here.
1569          * XXX: not sure we need this.
1570          */
1571         if ((mode & (VWRITE | VREAD | VEXEC)) &&
1572             va.va_type == VREG && MANDMODE(va.va_mode))
1573                 return (EACCES);

1575         /*
1576          * Access check is based on only
1577          * one of owner, group, public.
1578          * If not owner, then check group.
1579          * If not a member of the group,
1580          * then check public access.
1581          */
1582         if (crgetuid(cr) != va.va_uid) {
1583                 shift += 3;
1584                 if (!groupmember(va.va_gid, cr))
1585                         shift += 3;
1586         }

1588         /*
1589          * We need a vnode for secpolicy_vnode_access,
1590          * but the only thing it looks at is v_type,
1591          * so pass one of the templates above.
1592          */
1593         tvp = (va.va_type == VDIR) ?
1594             (vnode_t *)&tmpl_vdir :
1595             (vnode_t *)&tmpl_vreg;

1597         return (secpolicy_vnode_access2(cr, tvp, va.va_uid,
```

```
1598             va.va_mode << shift, mode));
1599 }

1601 /*
1602  * See smbfs_setattr
1603  */
1604 static int
1605 smbfs_accessx(void *arg, int mode, cred_t *cr)
1606 {
1607         vnode_t *vp = arg;
1608         /*
1609          * Note: The caller has checked the current zone,
1610          * the SMI_DEAD and VFS_UNMOUNTED flags, etc.
1611          */
1612         return (smbfs_access_rwx(vp->v_vfsp, vp->v_type, mode, cr));
1613 }

1615 /*
1616  * XXX
1617  * This op should support PSARC 2007/403, Modified Access Checks for CIFS
1618  */
1619 /* ARGSUSED */
1620 static int
1621 smbfs_access(vnode_t *vp, int mode, int flags, cred_t *cr, caller_context_t *ct)
1622 {
1623         vfs_t           *vfsp;
1624         smbmntinfo_t    *smi;

1626         vfsp = vp->v_vfsp;
1627         smi = VFTOSMI(vfsp);

1629         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1630                 return (EIO);

1632         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
1633                 return (EIO);

1635         return (smbfs_access_rwx(vfsp, vp->v_type, mode, cr));
1636 }


1639 /*
1640  * Flush local dirty pages to stable storage on the server.
1641  *
1642  * If FNODSYNC is specified, then there is nothing to do because
1643  * metadata changes are not cached on the client before being
1644  * sent to the server.
1645  */
1646 /* ARGSUSED */
1647 static int
1648 smbfs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
1649 {
1650         int             error = 0;
1651         smbmntinfo_t    *smi;
1652         smbnode_t       *np;
1653         struct smb_cred scred;

1655         np = VTOSMB(vp);
1656         smi = VTOSMI(vp);

1658         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1659                 return (EIO);

1661         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1662                 return (EIO);
```



```
1664         if ((syncflag & FNODSYNC) || IS_SWAPVP(vp))
1665                 return (0);

1667         if ((syncflag & (FSYNC|FDSYNC)) == 0)
1668                 return (0);

1670         /* Shared lock for n_fid use in _flush */
1671         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
1672                 return (EINTR);
1673         smb_credinit(&scred, cr);

1675         error = smbfs_smb_flush(np, &scred);

1677         smb_credrele(&scred);
1678         smbfs_rw_exit(&np->r_lkserlock);

1680         return (error);
1681 }

1683 /*
1684  * Last reference to vnode went away.
1685  */
1686 /* ARGSUSED */
1687 static void
1688 smbfs_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1689 {
1690         smbnode_t       *np;
1691         struct smb_cred scred;

1693         /*
1694          * Don't "bail out" for VFS_UNMOUNTED here,
1695          * as we want to do cleanup, etc.
1696          * See also pcfs_inactive
1697          */

1699         np = VTOSMB(vp);

1701         /*
1702          * If this is coming from the wrong zone, we let someone in the right
1703          * zone take care of it asynchronously.  We can get here due to
1704          * VN_RELE() being called from pageout() or fsflush().  This call may
1705          * potentially turn into an expensive no-op if, for instance, v_count
1706          * gets incremented in the meantime, but it's still correct.
1707          */

1709         /*
1710          * Defend against the possibility that higher-level callers
1711          * might not correctly balance open and close calls.  If we
1712          * get here with open references remaining, it means there
1713          * was a missing VOP_CLOSE somewhere.  If that happens, do
1714          * the close here so we don't "leak" FIDs on the server.
1715          *
1716          * Exclusive lock for modifying n_fid stuff.
1717          * Don't want this one ever interruptible.
1718          */
1719         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
1720         smb_credinit(&scred, cr);

1722         switch (np->n_ovtype) {
1723         case VNON:
1724                 /* not open (OK) */
1725                 break;

1727         case VDIR:
1728                 if (np->n_dirrefs == 0)
1729                         break;
```

```
1730                 SMBVDEBUG("open dir: refs %d path %s\n",
1731                     np->n_dirrefs, np->n_rpath);
1732                 /* Force last close. */
1733                 np->n_dirrefs = 1;
1734                 smbfs_rele_fid(np, &scred);
1735                 break;
1736 
1737         case VREG:
1738                 if (np->n_fidrefs == 0)
1739                         break;
1740                 SMBVDEBUG("open file: refs %d id 0x%x path %s\n",
1741                     np->n_fidrefs, np->n_fid, np->n_rpath);
1742                 /*
1743                  * Before otW close, make sure dirty pages written back.
1744                  */
1745                 if (vn_has_cached_data(vp)) {
1746                         /* smbfs_putapage() will acquire shared lock, so release
1747                          * exclusive lock temporally.
1748                          */
1749                         smbfs_rw_exit(&np->r_lkserlock);
1750 
1751                         (void) smbfs_putpage(vp, (offset_t) 0, 0, B_INVAL | B_AS
1752 
1753                         /* acquire exclusive lock again. */
1754                         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0
1755                 }
1756 #endif /* ! codereview */
1757                 /* Force last close. */
1758                 np->n_fidrefs = 1;
1759                 smbfs_rele_fid(np, &scred);
1760                 break;
1761 
1762         default:
1763                 SMBVDEBUG("bad n_ovtype %d\n", np->n_ovtype);
1764                 np->n_ovtype = VNON;
1765                 break;
1766         }
1767 
1768         smb_credrele(&scred);
1769         smbfs_rw_exit(&np->r_lkserlock);
1770 
1771         smbfs_addfree(np);
1772 }
1773 
1774 /*
1775  * Remote file system operations having to do with directory manipulation.
1776  */
1777 /* ARGSUSED */
1778 static int
1779 smbfs_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
1780         int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
1781         int *direntflags, pathname_t *realpnp)
1782 {
1783         vfs_t           *vfs;
1784         smbmntinfo_t    *smi;
1785         smbnode_t       *dnp;
1786         int             error;
1787 
1788         vfs = dvp->v_vfsp;
1789         smi = VFTOSMI(vfs);
1790 
1791         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1792                 return (EPERM);
1793 
1794         if (smi->smi_flags & SMI_DEAD || vfs->vfs_flag & VFS_UNMOUNTED)
1795                 return (EIO);
```



```
1797         dnp = VTOSMB(dvp);
1798 
1799         /*
1800          * Are we looking up extended attributes?  If so, "dvp" is
1801          * the file or directory for which we want attributes, and
1802          * we need a lookup of the (faked up) attribute directory
1803          * before we lookup the rest of the path.
1804          */
1805         if (flags & LOOKUP_XATTR) {
1806                 /*
1807                  * Require the xattr mount option.
1808                  */
1809                 if ((vfs->vfs_flag & VFS_XATTR) == 0)
1810                         return (EINVAL);
1811 
1812                 error = smbfs_get_xattrdir(dvp, vpp, cr, flags);
1813                 return (error);
1814         }
1815 
1816         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_READER, SMBINTR(dvp)))
1817                 return (EINTR);
1818 
1819         error = smbfslookup(dvp, nm, vpp, cr, 1, ct);
1820 
1821         smbfs_rw_exit(&dnp->r_rwlock);
1822 
1823         return (error);
1824 }
1825 
1826 /* ARGSUSED */
1827 static int
1828 smbfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
1829         int cache_ok, caller_context_t *ct)
1830 {
1831         int             error;
1832         int             supplen; /* supported length */
1833         vnode_t         *vp;
1834         smbnode_t       *np;
1835         smbnode_t       *dnp;
1836         smbmntinfo_t    *smi;
1837         /* struct smb_vc        *vcp; */
1838         const char      *ill;
1839         const char      *name = (const char *)nm;
1840         int             nmlen = strlen(nm);
1841         int             rplen;
1842         struct smb_cred scred;
1843         struct smbfattr fa;
1844 
1845         smi = VTOSMI(dvp);
1846         dnp = VTOSMB(dvp);
1847 
1848         ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);
1849 
1850 #ifdef NOT_YET
1851         vcp = SSTOVC(smi->smi_share);
1852 
1853         /* XXX: Should compute this once and store it in smbmntinfo_t */
1854         supplen = (SMB_DIALECT(vcp) >= SMB_DIALECT_LANMAN2_0) ? 255 : 12;
1855 #else
1856         supplen = 255;
1857 #endif
1858 
1859         /*
1860          * RWlock must be held, either reader or writer.
1861          * XXX: Can we check without looking directly
```

```
1862          * inside the struct smbfs_rwlock_t?
1863          */
1864         ASSERT(dnp->r_rwlock.count != 0);

1866         /*
1867          * If lookup is for "", just return dvp.
1868          * No need to perform any access checks.
1869          */
1870         if (nmlen == 0) {
1871                 VN_HOLD(dvp);
1872                 *vpp = dvp;
1873                 return (0);
1874         }

1876         /*
1877          * Can't do lookups in non-directories.
1878          */
1879         if (dvp->v_type != VDIR)
1880                 return (ENOTDIR);

1882         /*
1883          * Need search permission in the directory.
1884          */
1885         error = smbfs_access(dvp, VEXEC, 0, cr, ct);
1886         if (error)
1887                 return (error);

1889         /*
1890          * If lookup is for ".", just return dvp.
1891          * Access check was done above.
1892          */
1893         if (nmlen == 1 && name[0] == '.') {
1894                 VN_HOLD(dvp);
1895                 *vpp = dvp;
1896                 return (0);
1897         }

1899         /*
1900          * Now some sanity checks on the name.
1901          * First check the length.
1902          */
1903         if (nmlen > supplen)
1904                 return (ENAMETOOLONG);

1906         /*
1907          * Avoid surprises with characters that are
1908          * illegal in Windows file names.
1909          * Todo: CATIA mappings  XXX
1910          */
1911         ill = illegal_chars;
1912         if (dnp->n_flag & N_XATTR)
1913                 ill++; /* allow colon */
1914         if (strpbrk(nm, ill))
1915                 return (EINVAL);

1917         /*
1918          * Special handling for lookup of ".."
1919          *
1920          * We keep full pathnames (as seen on the server)
1921          * so we can just trim off the last component to
1922          * get the full pathname of the parent.  Note:
1923          * We don't actually copy and modify, but just
1924          * compute the trimmed length and pass that with
1925          * the current dir path (not null terminated).
1926          *
1927          * We don't go over-the-wire to get attributes
```



```
1928          * for ".." because we know it's a directory,
1929          * and we can just leave the rest "stale"
1930          * until someone does a getattr.
1931          */
1932         if (nmlen == 2 && name[0] == '.' && name[1] == '.') {
1933                 if (dvp->v_flag & VROOT) {
1934                         /*
1935                          * Already at the root.  This can happen
1936                          * with directory listings at the root,
1937                          * which lookup "." and ".." to get the
1938                          * inode numbers.  Let ".." be the same
1939                          * as "." in the FS root.
1940                          */
1941                         VN_HOLD(dvp);
1942                         *vpp = dvp;
1943                         return (0);
1944                 }

1946                 /*
1947                  * Special case for XATTR directory
1948                  */
1949                 if (dvp->v_flag & V_XATTRDIR) {
1950                         error = smbfs_xa_parent(dvp, vpp);
1951                         return (error);
1952                 }

1954                 /*
1955                  * Find the parent path length.
1956                  */
1957                 rplen = dnp->n_rplen;
1958                 ASSERT(rplen > 0);
1959                 while (--rplen >= 0) {
1960                         if (dnp->n_rpath[rplen] == '\\')
1961                                 break;
1962                 }
1963                 if (rplen <= 0) {
1964                         /* Found our way to the root. */
1965                         vp = SMBTOV(smi->smi_root);
1966                         VN_HOLD(vp);
1967                         *vpp = vp;
1968                         return (0);
1969                 }
1970                 np = smbfs_node_findcreate(smi,
1971                     dnp->n_rpath, rplen, NULL, 0, 0,
1972                     &smbfs_fattr0); /* force create */
1973                 ASSERT(np != NULL);
1974                 vp = SMBTOV(np);
1975                 vp->v_type = VDIR;

1977                 /* Success! */
1978                 *vpp = vp;
1979                 return (0);
1980         }

1982         /*
1983          * Normal lookup of a name under this directory.
1984          * Note we handled "", ".", ".." above.
1985          */
1986         if (cache_ok) {
1987                 /*
1988                  * The caller indicated that it's OK to use a
1989                  * cached result for this lookup, so try to
1990                  * reclaim a node from the smbfs node cache.
1991                  */
1992                 error = smbfslookup_cache(dvp, nm, nmlen, &vp, cr);
1993                 if (error)
```

```
1994                        return (error);
1995                if (vp != NULL) {
1996                        /* hold taken in lookup_cache */
1997                        *vpp = vp;
1998                        return (0);
1999                }
2000        }

2002        /*
2003         * OK, go over-the-wire to get the attributes,
2004         * then create the node.
2005         */
2006        smb_credinit(&scred, cr);
2007        /* Note: this can allocate a new "name" */
2008        error = smbfs_smb_lookup(dnp, &name, &nmlen, &fa, &scred);
2009        smb_credrele(&scred);
2010        if (error == ENOTDIR) {
2011                /*
2012                 * Lookup failed because this directory was
2013                 * removed or renamed by another client.
2014                 * Remove any cached attributes under it.
2015                 */
2016                smbfs_attrcache_remove(dnp);
2017                smbfs_attrcache_prune(dnp);
2018        }
2019        if (error)
2020                goto out;

2022        error = smbfs_nget(dvp, name, nmlen, &fa, &vp);
2023        if (error)
2024                goto out;

2026        /* Success! */
2027        *vpp = vp;

2029 out:
2030        /* smbfs_smb_lookup may have allocated name. */
2031        if (name != nm)
2032                smbfs_name_free(name, nmlen);

2034        return (error);
2035 }

2037 /*
2038  * smbfslookup_cache
2039  *
2040  * Try to reclaim a node from the smbfs node cache.
2041  * Some statistics for DEBUG.
2042  *
2043  * This mechanism lets us avoid many of the five (or more)
2044  * OtW lookup calls per file seen with "ls -l" if we search
2045  * the smbfs node cache for recently inactive(ated) nodes.
2046  */
2047 #ifdef DEBUG
2048 int smbfs_lookup_cache_calls = 0;
2049 int smbfs_lookup_cache_error = 0;
2050 int smbfs_lookup_cache_miss = 0;
2051 int smbfs_lookup_cache_stale = 0;
2052 int smbfs_lookup_cache_hits = 0;
2053 #endif /* DEBUG */

2055 /* ARGSUSED */
2056 static int
2057 smbfslookup_cache(vnode_t *dvp, char *nm, int nmlen,
2058         vnode_t **vpp, cred_t *cr)
2059 {
```



```
2060        struct vattr va;
2061        smbnode_t *dnp;
2062        smbnode_t *np;
2063        vnode_t *vp;
2064        int error;
2065        char sep;

2067        dnp = VTOSMB(dvp);
2068        *vpp = NULL;

2070 #ifdef DEBUG
2071        smbfs_lookup_cache_calls++;
2072 #endif

2074        /*
2075         * First make sure we can get attributes for the
2076         * directory.  Cached attributes are OK here.
2077         * If we removed or renamed the directory, this
2078         * will return ENOENT.  If someone else removed
2079         * this directory or file, we'll find out when we
2080         * try to open or get attributes.
2081         */
2082        va.va_mask = AT_TYPE | AT_MODE;
2083        error = smbfsgetattr(dvp, &va, cr);
2084        if (error) {
2085 #ifdef DEBUG
2086                smbfs_lookup_cache_error++;
2087 #endif
2088                return (error);
2089        }

2091        /*
2092         * Passing NULL smbfattr here so we will
2093         * just look, not create.
2094         */
2095        sep = SMBFS_DNP_SEP(dnp);
2096        np = smbfs_node_findcreate(dnp->n_mount,
2097            dnp->n_rpath, dnp->n_rplen,
2098            nm, nmlen, sep, NULL);
2099        if (np == NULL) {
2100 #ifdef DEBUG
2101                smbfs_lookup_cache_miss++;
2102 #endif
2103                return (0);
2104        }

2106        /*
2107         * Found it.  Attributes still valid?
2108         */
2109        vp = SMBTOV(np);
2110        if (np->r_attrtime <= gethrtime()) {
2111                /* stale */
2112 #ifdef DEBUG
2113                smbfs_lookup_cache_stale++;
2114 #endif
2115                VN_RELE(vp);
2116                return (0);
2117        }

2119        /*
2120         * Success!
2121         * Caller gets hold from smbfs_node_findcreate
2122         */
2123 #ifdef DEBUG
2124        smbfs_lookup_cache_hits++;
2125 #endif
```

```
2126         *vpp = vp;
2127         return (0);
2128 }

2130 /*
2131  * XXX
2132  * vsecattr_t is new to build 77, and we need to eventually support
2133  * it in order to create an ACL when an object is created.
2134  *
2135  * This op should support the new FIGNORECASE flag for case-insensitive
2136  * lookups, per PSARC 2007/244.
2137  */
2138 /* ARGSUSED */
2139 static int
2140 smbfs_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
2141         int mode, vnode_t **vpp, cred_t *cr, int lfaware, caller_context_t *ct,
2142         vsecattr_t *vsecp)
2143 {
2144         int             error;
2145         int             cerror;
2146         vfs_t           *vfsp;
2147         vnode_t         *vp;
2148 #ifdef NOT_YET
2149         smbnode_t       *np;
2150 #endif
2151         smbnode_t       *dnp;
2152         smbmntinfo_t    *smi;
2153         struct vattr    vattr;
2154         struct smbfattr fattr;
2155         struct smb_cred scred;
2156         const char *name = (const char *)nm;
2157         int             nmlen = strlen(nm);
2158         uint32_t        disp;
2159         uint16_t        fid;
2160         int             xattr;

2162         vfsp = dvp->v_vfsp;
2163         smi = VFTOSMI(vfsp);
2164         dnp = VTOSMB(dvp);
2165         vp = NULL;

2167         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2168                 return (EPERM);

2170         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
2171                 return (EIO);

2173         /*
2174          * Note: this may break mknod(2) calls to create a directory,
2175          * but that's obscure use.  Some other filesystems do this.
2176          * XXX: Later, redirect VDIR type here to _mkdir.
2177          */
2178         if (va->va_type != VREG)
2179                 return (EINVAL);

2181         /*
2182          * If the pathname is "", just use dvp, no checks.
2183          * Do this outside of the rwlock (like zfs).
2184          */
2185         if (nmlen == 0) {
2186                 VN_HOLD(dvp);
2187                 *vpp = dvp;
2188                 return (0);
2189         }

2191         /* Don't allow "." or ".." through here. */
```



```
2192         if ((nmlen == 1 && name[0] == '.') ||
2193             (nmlen == 2 && name[0] == '.' && name[1] == '.'))
2194                 return (EISDIR);

2196         /*
2197          * We make a copy of the attributes because the caller does not
2198          * expect us to change what va points to.
2199          */
2200         vattr = *va;

2202         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2203                 return (EINTR);
2204         smb_credinit(&scred, cr);

2206         /*
2207          * XXX: Do we need r_lkserlock too?
2208          * No use of any shared fid or fctx...
2209          */

2211         /*
2212          * NFS needs to go over the wire, just to be sure whether the
2213          * file exists or not.  Using a cached result is dangerous in
2214          * this case when making a decision regarding existence.
2215          *
2216          * The SMB protocol does NOT really need to go OTW here
2217          * thanks to the expressive NTCREATE disposition values.
2218          * Unfortunately, to do Unix access checks correctly,
2219          * we need to know if the object already exists.
2220          * When the object does not exist, we need VWRITE on
2221          * the directory.  Note: smbfslookup() checks VEXEC.
2222          */
2223         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2224         if (error == 0) {
2225                 /*
2226                  * The file already exists.  Error?
2227                  * NB: have a hold from smbfslookup
2228                  */
2229                 if (exclusive == EXCL) {
2230                         error = EEXIST;
2231                         VN_RELE(vp);
2232                         goto out;
2233                 }
2234                 /*
2235                  * Verify requested access.
2236                  */
2237                 error = smbfs_access(vp, mode, 0, cr, ct);
2238                 if (error) {
2239                         VN_RELE(vp);
2240                         goto out;
2241                 }

2243                 /*
2244                  * Truncate (if requested).
2245                  */
2246                 if ((vattr.va_mask & AT_SIZE) && vattr.va_size == 0) {
2247                         vattr.va_mask = AT_SIZE;
2248                         error = smbfssetattr(vp, &vattr, 0, cr);
2249                         if (error) {
2250                                 VN_RELE(vp);
2251                                 goto out;
2252                         }
2253                 }
2254                 /* Success! */
2255 #ifdef NOT_YET
2256                 vnevent_create(vp, ct);
2257 #endif
```

```
2258                *vpp = vp;
2259                goto out;
2260        }

2262        /*
2263         * The file did not exist.  Need VWRITE in the directory.
2264         */
2265        error = smbfs_access(dvp, VWRITE, 0, cr, ct);
2266        if (error)
2267                goto out;

2269        /*
2270         * Now things get tricky.  We also need to check the
2271         * requested open mode against the file we may create.
2272         * See comments at smbfs_access_rwx
2273         */
2274        error = smbfs_access_rwx(vfsp, VREG, mode, cr);
2275        if (error)
2276                goto out;

2278        /*
2279         * Now the code derived from Darwin,
2280         * but with greater use of NT_CREATE
2281         * disposition options.  Much changed.
2282         *
2283         * Create (or open) a new child node.
2284         * Note we handled "." and ".." above.
2285         */

2287        if (exclusive == EXCL)
2288                disp = NTCREATEX_DISP_CREATE;
2289        else {
2290                /* Truncate regular files if requested. */
2291                if ((va->va_type == VREG) &&
2292                    (va->va_mask & AT_SIZE) &&
2293                    (va->va_size == 0))
2294                        disp = NTCREATEX_DISP_OVERWRITE_IF;
2295                else
2296                        disp = NTCREATEX_DISP_OPEN_IF;
2297        }
2298        xattr = (dnp->n_flag & N_XATTR) ? 1 : 0;
2299        error = smbfs_smb_create(dnp,
2300            name, nmlen, xattr,
2301            disp, &scred, &fid);
2302        if (error)
2303                goto out;

2305        /*
2306         * XXX: Missing some code here to deal with
2307         * the case where we opened an existing file,
2308         * it's size is larger than 32-bits, and we're
2309         * setting the size from a process that's not
2310         * aware of large file offsets.  i.e.
2311         * from the NFS3 code:
2312         */
2313 #if NOT_YET /* XXX */
2314        if ((vattr.va_mask & AT_SIZE) &&
2315            vp->v_type == VREG) {
2316                np = VTOSMB(vp);
2317                /*
2318                 * Check here for large file handled
2319                 * by LF-unaware process (as
2320                 * ufs_create() does)
2321                 */
2322                if (!(lfaware & FOFFMAX)) {
2323                        mutex_enter(&np->r_statelock);
```



```
2324                        if (np->r_size > MAXOFF32_T)
2325                                error = EOVERFLOW;
2326                        mutex_exit(&np->r_statelock);
2327                }
2328                if (!error) {
2329                        vattr.va_mask = AT_SIZE;
2330                        error = smbfssetattr(vp,
2331                            &vattr, 0, cr);
2332                }
2333        }
2334 #endif /* XXX */
2335        /*
2336         * Should use the fid to get/set the size
2337         * while we have it opened here.  See above.
2338         */

2340        cerror = smbfs_smb_close(smi->smi_share, fid, NULL, &scred);
2341        if (cerror)
2342                SMBVDEBUG("error %d closing %s\\%s\n",
2343                    cerror, dnp->n_rpath, name);

2345        /*
2346         * In the open case, the name may differ a little
2347         * from what we passed to create (case, etc.)
2348         * so call lookup to get the (opened) name.
2349         *
2350         * XXX: Could avoid this extra lookup if the
2351         * "createact" result from NT_CREATE says we
2352         * created the object.
2353         */
2354        error = smbfs_smb_lookup(dnp, &name, &nmlen, &fattr, &scred);
2355        if (error)
2356                goto out;

2358        /* update attr and directory cache */
2359        smbfs_attr_touchdir(dnp);

2361        error = smbfs_nget(dvp, name, nmlen, &fattr, &vp);
2362        if (error)
2363                goto out;

2365        /* XXX invalidate pages if we truncated? */

2367        /* Success! */
2368        *vpp = vp;
2369        error = 0;

2371 out:
2372        smb_credrele(&scred);
2373        smbfs_rw_exit(&dnp->r_rwlock);
2374        if (name != nm)
2375                smbfs_name_free(name, nmlen);
2376        return (error);
2377 }

2379 /*
2380  * XXX
2381  * This op should support the new FIGNORECASE flag for case-insensitive
2382  * lookups, per PSARC 2007/244.
2383  */
2384 /* ARGSUSED */
2385 static int
2386 smbfs_remove(vnode_t *dvp, char *nm, cred_t *cr, caller_context_t *ct,
2387        int flags)
2388 {
2389        int             error;
```

```
2390         vnode_t         *vp;
2391         smbnode_t       *np;
2392         smbnode_t       *dnp;
2393         struct smb_cred scred;
2394         /* enum smbfsstat status; */
2395         smbmntinfo_t    *smi;

2397         smi = VTOSMI(dvp);

2399         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2400                 return (EPERM);

2402         if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2403                 return (EIO);

2405         dnp = VTOSMB(dvp);
2406         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2407                 return (EINTR);
2408         smb_credinit(&scred, cr);

2410         /*
2411          * Verify access to the dirctory.
2412          */
2413         error = smbfs_access(dvp, VWRITE|VEXEC, 0, cr, ct);
2414         if (error)
2415                 goto out;

2417         /*
2418          * NOTE:  the darwin code gets the "vp" passed in so it looks
2419          * like the "vp" has probably been "lookup"ed by the VFS layer.
2420          * It looks like we will need to lookup the vp to check the
2421          * caches and check if the object being deleted is a directory.
2422          */
2423         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2424         if (error)
2425                 goto out;

2427         /* Never allow link/unlink directories on CIFS. */
2428         if (vp->v_type == VDIR) {
2429                 VN_RELE(vp);
2430                 error = EPERM;
2431                 goto out;
2432         }

2434         /*
2435          * Now we have the real reference count on the vnode
2436          * Do we have the file open?
2437          */
2438         np = VTOSMB(vp);
2439         mutex_enter(&np->r_statelock);
2440         if ((vp->v_count > 1) && (np->n_fidrefs > 0)) {
2441                 /*
2442                  * NFS does a rename on remove here.
2443                  * Probably not applicable for SMB.
2444                  * Like Darwin, just return EBUSY.
2445                  *
2446                  * XXX: Todo - Use Trans2rename, and
2447                  * if that fails, ask the server to
2448                  * set the delete-on-close flag.
2449                  */
2450                 mutex_exit(&np->r_statelock);
2451                 error = EBUSY;
2452         } else {
2453                 smbfs_attrcache_rm_locked(np);
2454                 mutex_exit(&np->r_statelock);
```



```
2456                 error = smbfs_smb_delete(np, &scred, NULL, 0, 0);

2458                 /*
2459                  * If the file should no longer exist, discard
2460                  * any cached attributes under this node.
2461                  */
2462                 switch (error) {
2463                 case 0:
2464                 case ENOENT:
2465                 case ENOTDIR:
2466                         smbfs_attrcache_prune(np);
2467                         break;
2468                 }
2469         }

2471         VN_RELE(vp);

2473 out:
2474         smb_credrele(&scred);
2475         smbfs_rw_exit(&dnp->r_rwlock);

2477         return (error);
2478 }


2481 /*
2482  * XXX
2483  * This op should support the new FIGNORECASE flag for case-insensitive
2484  * lookups, per PSARC 2007/244.
2485  */
2486 /* ARGSUSED */
2487 static int
2488 smbfs_rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
2489         caller_context_t *ct, int flags)
2490 {
2491         /* vnode_t              *realvp; */

2493         if (curproc->p_zone != VTOSMI(odvp)->smi_zone_ref.zref_zone ||
2494             curproc->p_zone != VTOSMI(ndvp)->smi_zone_ref.zref_zone)
2495                 return (EPERM);

2497         if (VTOSMI(odvp)->smi_flags & SMI_DEAD ||
2498             VTOSMI(ndvp)->smi_flags & SMI_DEAD ||
2499             odvp->v_vfsp->vfs_flag & VFS_UNMOUNTED ||
2500             ndvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2501                 return (EIO);

2503         return (smbfsrename(odvp, onm, ndvp, nnm, cr, ct));
2504 }

2506 /*
2507  * smbfsrename does the real work of renaming in SMBFS
2508  */
2509 /* ARGSUSED */
2510 static int
2511 smbfsrename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
2512         caller_context_t *ct)
2513 {
2514         int             error;
2515         int             nvp_locked = 0;
2516         vnode_t         *nvp = NULL;
2517         vnode_t         *ovp = NULL;
2518         smbnode_t       *onp;
2519         smbnode_t       *nnp;
2520         smbnode_t       *odnp;
2521         smbnode_t       *ndnp;
```

```
2522         struct smb_cred scred;
2523         /* enum smbfsstat       status; */

2525         ASSERT(curproc->p_zone == VTOSMI(odvp)->smi_zone_ref.zref_zone);

2527         if (strcmp(onm, ".") == 0 || strcmp(onm, "..") == 0 ||
2528             strcmp(nnm, ".") == 0 || strcmp(nnm, "..") == 0)
2529                 return (EINVAL);

2531         /*
2532          * Check that everything is on the same filesystem.
2533          * vn_rename checks the fsid's, but in case we don't
2534          * fill those in correctly, check here too.
2535          */
2536         if (odvp->v_vfsp != ndvp->v_vfsp)
2537                 return (EXDEV);

2539         odnp = VTOSMB(odvp);
2540         ndnp = VTOSMB(ndvp);

2542         /*
2543          * Avoid deadlock here on old vs new directory nodes
2544          * by always taking the locks in order of address.
2545          * The order is arbitrary, but must be consistent.
2546          */
2547         if (odnp < ndnp) {
2548                 if (smbfs_rw_enter_sig(&odnp->r_rwlock, RW_WRITER,
2549                     SMBINTR(odvp)))
2550                         return (EINTR);
2551                 if (smbfs_rw_enter_sig(&ndnp->r_rwlock, RW_WRITER,
2552                     SMBINTR(ndvp))) {
2553                         smbfs_rw_exit(&odnp->r_rwlock);
2554                         return (EINTR);
2555                 }
2556         } else {
2557                 if (smbfs_rw_enter_sig(&ndnp->r_rwlock, RW_WRITER,
2558                     SMBINTR(ndvp)))
2559                         return (EINTR);
2560                 if (smbfs_rw_enter_sig(&odnp->r_rwlock, RW_WRITER,
2561                     SMBINTR(odvp))) {
2562                         smbfs_rw_exit(&ndnp->r_rwlock);
2563                         return (EINTR);
2564                 }
2565         }
2566         smb_credinit(&scred, cr);
2567         /*
2568          * No returns after this point (goto out)
2569          */

2571         /*
2572          * Need write access on source and target.
2573          * Server takes care of most checks.
2574          */
2575         error = smbfs_access(odvp, VWRITE|VEXEC, 0, cr, ct);
2576         if (error)
2577                 goto out;
2578         if (odvp != ndvp) {
2579                 error = smbfs_access(ndvp, VWRITE, 0, cr, ct);
2580                 if (error)
2581                         goto out;
2582         }

2584         /*
2585          * Lookup the source name.  Must already exist.
2586          */
2587         error = smbfslookup(odvp, onm, &ovp, cr, 0, ct);
```



```
2588         if (error)
2589                 goto out;

2591         /*
2592          * Lookup the target file.  If it exists, it needs to be
2593          * checked to see whether it is a mount point and whether
2594          * it is active (open).
2595          */
2596         error = smbfslookup(ndvp, nnm, &nvp, cr, 0, ct);
2597         if (!error) {
2598                 /*
2599                  * Target (nvp) already exists.  Check that it
2600                  * has the same type as the source.  The server
2601                  * will check this also, (and more reliably) but
2602                  * this lets us return the correct error codes.
2603                  */
2604                 if (ovp->v_type == VDIR) {
2605                         if (nvp->v_type != VDIR) {
2606                                 error = ENOTDIR;
2607                                 goto out;
2608                         }
2609                 } else {
2610                         if (nvp->v_type == VDIR) {
2611                                 error = EISDIR;
2612                                 goto out;
2613                         }
2614                 }

2616                 /*
2617                  * POSIX dictates that when the source and target
2618                  * entries refer to the same file object, rename
2619                  * must do nothing and exit without error.
2620                  */
2621                 if (ovp == nvp) {
2622                         error = 0;
2623                         goto out;
2624                 }

2626                 /*
2627                  * Also must ensure the target is not a mount point,
2628                  * and keep mount/umount away until we're done.
2629                  */
2630                 if (vn_vfsrlock(nvp)) {
2631                         error = EBUSY;
2632                         goto out;
2633                 }
2634                 nvp_locked = 1;
2635                 if (vn_mountedvfs(nvp) != NULL) {
2636                         error = EBUSY;
2637                         goto out;
2638                 }

2640                 /*
2641                  * CIFS gives a SHARING_VIOLATION error when
2642                  * trying to rename onto an exising object,
2643                  * so try to remove the target first.
2644                  * (Only for files, not directories.)
2645                  */
2646                 if (nvp->v_type == VDIR) {
2647                         error = EEXIST;
2648                         goto out;
2649                 }

2651                 /*
2652                  * Nodes that are "not active" here have v_count=2
2653                  * because vn_renameat (our caller) did a lookup on
```

```
2654                  * both the source and target before this call.
2655                  * Otherwise this similar to smbfs_remove.
2656                  */
2657                 nnp = VTOSMB(nvp);
2658                 mutex_enter(&nnp->r_statelock);
2659                 if ((nvp->v_count > 2) && (nnp->n_fidrefs > 0)) {
2660                         /*
2661                          * The target file exists, is not the same as
2662                          * the source file, and is active.  Other FS
2663                          * implementations unlink the target here.
2664                          * For SMB, we don't assume we can remove an
2665                          * open file.  Return an error instead.
2666                          */
2667                         mutex_exit(&nnp->r_statelock);
2668                         error = EBUSY;
2669                         goto out;
2670                 }

2672                 /*
2673                  * Target file is not active. Try to remove it.
2674                  */
2675                 smbfs_attrcache_rm_locked(nnp);
2676                 mutex_exit(&nnp->r_statelock);

2678                 error = smbfs_smb_delete(nnp, &scred, NULL, 0, 0);

2680                 /*
2681                  * Similar to smbfs_remove
2682                  */
2683                 switch (error) {
2684                 case 0:
2685                 case ENOENT:
2686                 case ENOTDIR:
2687                         smbfs_attrcache_prune(nnp);
2688                         break;
2689                 }

2691                 if (error)
2692                         goto out;
2693                 /*
2694                  * OK, removed the target file.  Continue as if
2695                  * lookup target had failed (nvp == NULL).
2696                  */
2697                 vn_vfsunlock(nvp);
2698                 nvp_locked = 0;
2699                 VN_RELE(nvp);
2700                 nvp = NULL;
2701         } /* nvp */

2703         onp = VTOSMB(ovp);
2704         smbfs_attrcache_remove(onp);

2706         error = smbfs_smb_rename(onp, ndnp, nnm, strlen(nnm), &scred);

2708         /*
2709          * If the old name should no longer exist,
2710          * discard any cached attributes under it.
2711          */
2712         if (error == 0)
2713                 smbfs_attrcache_prune(onp);

2715 out:
2716         if (nvp) {
2717                 if (nvp_locked)
2718                         vn_vfsunlock(nvp);
2719                 VN_RELE(nvp);
```



```
2720         }
2721         if (ovp)
2722                 VN_RELE(ovp);

2724         smb_credrele(&scred);
2725         smbfs_rw_exit(&odnp->r_rwlock);
2726         smbfs_rw_exit(&ndnp->r_rwlock);

2728         return (error);
2729 }

2731 /*
2732  * XXX
2733  * vsecattr_t is new to build 77, and we need to eventually support
2734  * it in order to create an ACL when an object is created.
2735  *
2736  * This op should support the new FIGNORECASE flag for case-insensitive
2737  * lookups, per PSARC 2007/244.
2738  */
2739 /* ARGSUSED */
2740 static int
2741 smbfs_mkdir(vnode_t *dvp, char *nm, struct vattr *va, vnode_t **vpp,
2742         cred_t *cr, caller_context_t *ct, int flags, vsecattr_t *vsecp)
2743 {
2744         vnode_t         *vp;
2745         struct smbnode  *dnp = VTOSMB(dvp);
2746         struct smbmntinfo *smi = VTOSMI(dvp);
2747         struct smb_cred scred;
2748         struct smbfattr fattr;
2749         const char              *name = (const char *) nm;
2750         int             nmlen = strlen(name);
2751         int             error, hiderr;

2753         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2754                 return (EPERM);

2756         if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2757                 return (EIO);

2759         if ((nmlen == 1 && name[0] == '.') ||
2760             (nmlen == 2 && name[0] == '.' && name[1] == '.'))
2761                 return (EEXIST);

2763         /* Only plain files are allowed in V_XATTRDIR. */
2764         if (dvp->v_flag & V_XATTRDIR)
2765                 return (EINVAL);

2767         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2768                 return (EINTR);
2769         smb_credinit(&scred, cr);

2771         /*
2772          * XXX: Do we need r_lkserlock too?
2773          * No use of any shared fid or fctx...
2774          */

2776         /*
2777          * Require write access in the containing directory.
2778          */
2779         error = smbfs_access(dvp, VWRITE, 0, cr, ct);
2780         if (error)
2781                 goto out;

2783         error = smbfs_smb_mkdir(dnp, name, nmlen, &scred);
2784         if (error)
2785                 goto out;
```

```
2787         error = smbfs_smb_lookup(dnp, &name, &nmlen, &fattr, &scred);
2788         if (error)
2789                 goto out;

2791         smbfs_attr_touchdir(dnp);

2793         error = smbfs_nget(dvp, name, nmlen, &fattr, &vp);
2794         if (error)
2795                 goto out;

2797         if (name[0] == '.')
2798                 if ((hiderr = smbfs_smb_hideit(VTOSMB(vp), NULL, 0, &scred)))
2799                         SMBVDEBUG("hide failure %d\n", hiderr);

2801         /* Success! */
2802         *vpp = vp;
2803         error = 0;
2804 out:
2805         smb_credrele(&scred);
2806         smbfs_rw_exit(&dnp->r_rwlock);

2808         if (name != nm)
2809                 smbfs_name_free(name, nmlen);

2811         return (error);
2812 }

2814 /*
2815  * XXX
2816  * This op should support the new FIGNORECASE flag for case-insensitive
2817  * lookups, per PSARC 2007/244.
2818  */
2819 /* ARGSUSED */
2820 static int
2821 smbfs_rmdir(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
2822         caller_context_t *ct, int flags)
2823 {
2824         vnode_t         *vp = NULL;
2825         int             vp_locked = 0;
2826         struct smbmntinfo *smi = VTOSMI(dvp);
2827         struct smbnode  *dnp = VTOSMB(dvp);
2828         struct smbnode  *np;
2829         struct smb_cred scred;
2830         int             error;

2832         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2833                 return (EPERM);

2835         if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2836                 return (EIO);

2838         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2839                 return (EINTR);
2840         smb_credinit(&scred, cr);

2842         /*
2843          * Require w/x access in the containing directory.
2844          * Server handles all other access checks.
2845          */
2846         error = smbfs_access(dvp, VEXEC|VWRITE, 0, cr, ct);
2847         if (error)
2848                 goto out;

2850         /*
2851          * First lookup the entry to be removed.
```



```
2852          */
2853         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2854         if (error)
2855                 goto out;
2856         np = VTOSMB(vp);

2858         /*
2859          * Disallow rmdir of "." or current dir, or the FS root.
2860          * Also make sure it's a directory, not a mount point,
2861          * and lock to keep mount/umount away until we're done.
2862          */
2863         if ((vp == dvp) || (vp == cdir) || (vp->v_flag & VROOT)) {
2864                 error = EINVAL;
2865                 goto out;
2866         }
2867         if (vp->v_type != VDIR) {
2868                 error = ENOTDIR;
2869                 goto out;
2870         }
2871         if (vn_vfsrlock(vp)) {
2872                 error = EBUSY;
2873                 goto out;
2874         }
2875         vp_locked = 1;
2876         if (vn_mountedvfs(vp) != NULL) {
2877                 error = EBUSY;
2878                 goto out;
2879         }

2881         smbfs_attrcache_remove(np);
2882         error = smbfs_smb_rmdir(np, &scred);

2884         /*
2885          * Similar to smbfs_remove
2886          */
2887         switch (error) {
2888         case 0:
2889         case ENOENT:
2890         case ENOTDIR:
2891                 smbfs_attrcache_prune(np);
2892                 break;
2893         }

2895         if (error)
2896                 goto out;

2898         mutex_enter(&np->r_statelock);
2899         dnp->n_flag |= NMODIFIED;
2900         mutex_exit(&np->r_statelock);
2901         smbfs_attr_touchdir(dnp);
2902         smbfs_rmhash(np);

2904 out:
2905         if (vp) {
2906                 if (vp_locked)
2907                         vn_vfsunlock(vp);
2908                 VN_RELE(vp);
2909         }
2910         smb_credrele(&scred);
2911         smbfs_rw_exit(&dnp->r_rwlock);

2913         return (error);
2914 }


2917 /* ARGSUSED */
```

```
2918 static int
2919 smbfs_readdir(vnode_t *vp, struct uio *uiop, cred_t *cr, int *eofp,
2920         caller_context_t *ct, int flags)
2921 {
2922         struct smbnode  *np = VTOSMB(vp);
2923         int             error = 0;
2924         smbmntinfo_t    *smi;

2926         smi = VTOSMI(vp);

2928         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2929                 return (EIO);

2931         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2932                 return (EIO);

2934         /*
2935          * Require read access in the directory.
2936          */
2937         error = smbfs_access(vp, VREAD, 0, cr, ct);
2938         if (error)
2939                 return (error);

2941         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_READER));

2943         /*
2944          * XXX: Todo readdir cache here
2945          * Note: NFS code is just below this.
2946          *
2947          * I am serializing the entire readdir opreation
2948          * now since we have not yet implemented readdir
2949          * cache. This fix needs to be revisited once
2950          * we implement readdir cache.
2951          */
2952         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
2953                 return (EINTR);

2955         error = smbfs_readvdir(vp, uiop, cr, eofp, ct);

2957         smbfs_rw_exit(&np->r_lkserlock);

2959         return (error);
2960 }

2962 /* ARGSUSED */
2963 static int
2964 smbfs_readvdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
2965         caller_context_t *ct)
2966 {
2967         /*
2968          * Note: "limit" tells the SMB-level FindFirst/FindNext
2969          * functions how many directory entries to request in
2970          * each OtW call.  It needs to be large enough so that
2971          * we don't make lots of tiny OtW requests, but there's
2972          * no point making it larger than the maximum number of
2973          * OtW entries that would fit in a maximum sized trans2
2974          * response (64k / 48).  Beyond that, it's just tuning.
2975          * WinNT used 512, Win2k used 1366.  We use 1000.
2976          */
2977         static const int limit = 1000;
2978         /* Largest possible dirent size. */
2979         static const size_t dbufsiz = DIRENT64_RECLEN(SMB_MAXFNAMELEN);
2980         struct smb_cred scred;
2981         vnode_t         *newvp;
2982         struct smbnode  *np = VTOSMB(vp);
2983         struct smbfs_fctx *ctx;
```



```
2984         struct dirent64 *dp;
2985         ssize_t         save_resid;
2986         offset_t        save_offset; /* 64 bits */
2987         int             offset; /* yes, 32 bits */
2988         int             nmlen, error;
2989         ushort_t        reclen;

2991         ASSERT(curproc->p_zone == VTOSMI(vp)->smi_zone_ref.zref_zone);

2993         /* Make sure we serialize for n_dirseq use. */
2994         ASSERT(smbfs_rw_lock_held(&np->r_lkserlock, RW_WRITER));

2996         /*
2997          * Make sure smbfs_open filled in n_dirseq
2998          */
2999         if (np->n_dirseq == NULL)
3000                 return (EBADF);

3002         /* Check for overflow of (32-bit) directory offset. */
3003         if (uio->uio_loffset < 0 || uio->uio_loffset > INT32_MAX ||
3004             (uio->uio_loffset + uio->uio_resid) > INT32_MAX)
3005                 return (EINVAL);

3007         /* Require space for at least one dirent. */
3008         if (uio->uio_resid < dbufsiz)
3009                 return (EINVAL);

3011         SMBVDEBUG("dirname='%s'\n", np->n_rpath);
3012         smb_credinit(&scred, cr);
3013         dp = kmem_alloc(dbufsiz, KM_SLEEP);

3015         save_resid = uio->uio_resid;
3016         save_offset = uio->uio_loffset;
3017         offset = uio->uio_offset;
3018         SMBVDEBUG("in: offset=%d, resid=%d\n",
3019             (int)uio->uio_offset, (int)uio->uio_resid);
3020         error = 0;

3022         /*
3023          * Generate the "." and ".." entries here so we can
3024          * (1) make sure they appear (but only once), and
3025          * (2) deal with getting their I numbers which the
3026          * findnext below does only for normal names.
3027          */
3028         while (offset < FIRST_DIROFS) {
3029                 /*
3030                  * Tricky bit filling in the first two:
3031                  * offset 0 is ".", offset 1 is ".."
3032                  * so strlen of these is offset+1.
3033                  */
3034                 reclen = DIRENT64_RECLEN(offset + 1);
3035                 if (uio->uio_resid < reclen)
3036                         goto out;
3037                 bzero(dp, reclen);
3038                 dp->d_reclen = reclen;
3039                 dp->d_name[0] = '.';
3040                 dp->d_name[1] = '.';
3041                 dp->d_name[offset + 1] = '\0';
3042                 /*
3043                  * Want the real I-numbers for the "." and ".."
3044                  * entries.  For these two names, we know that
3045                  * smbfslookup can get the nodes efficiently.
3046                  */
3047                 error = smbfslookup(vp, dp->d_name, &newvp, cr, 1, ct);
3048                 if (error) {
3049                         dp->d_ino = np->n_ino + offset; /* fiction */
```

```
3050                } else {
3051                        dp->d_ino = VTOSMB(newvp)->n_ino;
3052                        VN_RELE(newvp);
3053                }
3054                /*
3055                 * Note: d_off is the offset that a user-level program
3056                 * should seek to for reading the NEXT directory entry.
3057                 * See libc: readdir, telldir, seekdir
3058                 */
3059                dp->d_off = offset + 1;
3060                error = uiomove(dp, reclen, UIO_READ, uio);
3061                if (error)
3062                        goto out;
3063                /*
3064                 * Note: uiomove updates uio->uio_offset,
3065                 * but we want it to be our "cookie" value,
3066                 * which just counts dirents ignoring size.
3067                 */
3068                uio->uio_offset = ++offset;
3069        }

3071        /*
3072         * If there was a backward seek, we have to reopen.
3073         */
3074        if (offset < np->n_dirofs) {
3075                SMBVDEBUG("Reopening search %d:%d\n",
3076                    offset, np->n_dirofs);
3077                error = smbfs_smb_findopen(np, "*", 1,
3078                    SMB_FA_SYSTEM | SMB_FA_HIDDEN | SMB_FA_DIR,
3079                    &scred, &ctx);
3080                if (error) {
3081                        SMBVDEBUG("can not open search, error = %d", error);
3082                        goto out;
3083                }
3084                /* free the old one */
3085                (void) smbfs_smb_findclose(np->n_dirseq, &scred);
3086                /* save the new one */
3087                np->n_dirseq = ctx;
3088                np->n_dirofs = FIRST_DIROFS;
3089        } else {
3090                ctx = np->n_dirseq;
3091        }

3093        /*
3094         * Skip entries before the requested offset.
3095         */
3096        while (np->n_dirofs < offset) {
3097                error = smbfs_smb_findnext(ctx, limit, &scred);
3098                if (error != 0)
3099                        goto out;
3100                np->n_dirofs++;
3101        }

3103        /*
3104         * While there's room in the caller's buffer:
3105         *      get a directory entry from SMB,
3106         *      convert to a dirent, copyout.
3107         * We stop when there is no longer room for a
3108         * maximum sized dirent because we must decide
3109         * before we know anything about the next entry.
3110         */
3111        while (uio->uio_resid >= dbufsiz) {
3112                error = smbfs_smb_findnext(ctx, limit, &scred);
3113                if (error != 0)
3114                        goto out;
3115                np->n_dirofs++;
```



```
3117                /* Sanity check the name length. */
3118                nmlen = ctx->f_nmlen;
3119                if (nmlen > SMB_MAXFNAMELEN) {
3120                        nmlen = SMB_MAXFNAMELEN;
3121                        SMBVDEBUG("Truncating name: %s\n", ctx->f_name);
3122                }
3123                if (smbfs_fastlookup) {
3124                        /* See comment at smbfs_fastlookup above. */
3125                        if (smbfs_nget(vp, ctx->f_name, nmlen,
3126                            &ctx->f_attr, &newvp) == 0)
3127                                VN_RELE(newvp);
3128                }

3130                reclen = DIRENT64_RECLEN(nmlen);
3131                bzero(dp, reclen);
3132                dp->d_reclen = reclen;
3133                bcopy(ctx->f_name, dp->d_name, nmlen);
3134                dp->d_name[nmlen] = '\0';
3135                dp->d_ino = ctx->f_inum;
3136                dp->d_off = offset + 1; /* See d_off comment above */
3137                error = uiomove(dp, reclen, UIO_READ, uio);
3138                if (error)
3139                        goto out;
3140                /* See comment re. uio_offset above. */
3141                uio->uio_offset = ++offset;
3142        }

3144 out:
3145        /*
3146         * When we come to the end of a directory, the
3147         * SMB-level functions return ENOENT, but the
3148         * caller is not expecting an error return.
3149         *
3150         * Also note that we must delay the call to
3151         * smbfs_smb_findclose(np->n_dirseq, ...)
3152         * until smbfs_close so that all reads at the
3153         * end of the directory will return no data.
3154         */
3155        if (error == ENOENT) {
3156                error = 0;
3157                if (eofp)
3158                        *eofp = 1;
3159        }
3160        /*
3161         * If we encountered an error (i.e. "access denied")
3162         * from the FindFirst call, we will have copied out
3163         * the "." and ".." entries leaving offset == 2.
3164         * In that case, restore the original offset/resid
3165         * so the caller gets no data with the error.
3166         */
3167        if (error != 0 && offset == FIRST_DIROFS) {
3168                uio->uio_loffset = save_offset;
3169                uio->uio_resid = save_resid;
3170        }
3171        SMBVDEBUG("out: offset=%d, resid=%d\n",
3172            (int)uio->uio_offset, (int)uio->uio_resid);

3174        kmem_free(dp, dbufsiz);
3175        smb_credrele(&scred);
3176        return (error);
3177 }


3180 /*
3181  * The pair of functions VOP_RWLOCK, VOP_RWUNLOCK
```

```
3182  * are optional functions that are called by:
3183  *    getdents, before/after VOP_READDIR
3184  *    pread, before/after ... VOP_READ
3185  *    pwrite, before/after ... VOP_WRITE
3186  *    (other places)
3187  *
3188  * Careful here: None of the above check for any
3189  * error returns from VOP_RWLOCK / VOP_RWUNLOCK!
3190  * In fact, the return value from _rwlock is NOT
3191  * an error code, but V_WRITELOCK_TRUE / _FALSE.
3192  *
3193  * Therefore, it's up to _this_ code to make sure
3194  * the lock state remains balanced, which means
3195  * we can't "bail out" on interrupts, etc.
3196  */

3198 /* ARGSUSED2 */
3199 static int
3200 smbfs_rwlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
3201 {
3202         smbnode_t       *np = VTOSMB(vp);

3204         if (!write_lock) {
3205                 (void) smbfs_rw_enter_sig(&np->r_rwlock, RW_READER, FALSE);
3206                 return (V_WRITELOCK_FALSE);
3207         }


3210         (void) smbfs_rw_enter_sig(&np->r_rwlock, RW_WRITER, FALSE);
3211         return (V_WRITELOCK_TRUE);
3212 }

3214 /* ARGSUSED */
3215 static void
3216 smbfs_rwunlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
3217 {
3218         smbnode_t       *np = VTOSMB(vp);

3220         smbfs_rw_exit(&np->r_rwlock);
3221 }


3224 /* ARGSUSED */
3225 static int
3226 smbfs_seek(vnode_t *vp, offset_t ooff, offset_t *noffp, caller_context_t *ct)
3227 {
3228         smbmntinfo_t    *smi;

3230         smi = VTOSMI(vp);

3232         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3233                 return (EPERM);

3235         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3236                 return (EIO);

3238         /*
3239          * Because we stuff the readdir cookie into the offset field
3240          * someone may attempt to do an lseek with the cookie which
3241          * we want to succeed.
3242          */
3243         if (vp->v_type == VDIR)
3244                 return (0);

3246         /* Like NFS3, just check for 63-bit overflow. */
3247         if (*noffp < 0)
```



```
3248                 return (EINVAL);

3250         return (0);
3251 }


3254 /*
3255  * XXX
3256  * This op may need to support PSARC 2007/440, nbmand changes for CIFS Service.
3257  */
3258 static int
3259 smbfs_frlock(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
3260         offset_t offset, struct flk_callback *flk_cbp, cred_t *cr,
3261         caller_context_t *ct)
3262 {
3263         if (curproc->p_zone != VTOSMI(vp)->smi_zone_ref.zref_zone)
3264                 return (EIO);

3266         if (VTOSMI(vp)->smi_flags & SMI_LLOCK)
3267                 return (fs_frlock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
3268         else
3269                 return (ENOSYS);
3270 }

3272 /*
3273  * Free storage space associated with the specified vnode.  The portion
3274  * to be freed is specified by bfp->l_start and bfp->l_len (already
3275  * normalized to a "whence" of 0).
3276  *
3277  * Called by fcntl(fd, F_FREESP, lkp) for libc:ftruncate, etc.
3278  */
3279 /* ARGSUSED */
3280 static int
3281 smbfs_space(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
3282         offset_t offset, cred_t *cr, caller_context_t *ct)
3283 {
3284         int             error;
3285         smbmntinfo_t    *smi;

3287         smi = VTOSMI(vp);

3289         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3290                 return (EIO);

3292         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3293                 return (EIO);

3295         /* Caller (fcntl) has checked v_type */
3296         ASSERT(vp->v_type == VREG);
3297         if (cmd != F_FREESP)
3298                 return (EINVAL);

3300         /*
3301          * Like NFS3, no 32-bit offset checks here.
3302          * Our SMB layer takes care to return EFBIG
3303          * when it has to fallback to a 32-bit call.
3304          */

3306         error = convoff(vp, bfp, 0, offset);
3307         if (!error) {
3308                 ASSERT(bfp->l_start >= 0);
3309                 if (bfp->l_len == 0) {
3310                         struct vattr va;

3312                         /*
3313                          * ftruncate should not change the ctime and
```

```
3314                         * mtime if we truncate the file to its
3315                         * previous size.
3316                         */
3317                        va.va_mask = AT_SIZE;
3318                        error = smbfsgetattr(vp, &va, cr);
3319                        if (error || va.va_size == bfp->l_start)
3320                                return (error);
3321                        va.va_mask = AT_SIZE;
3322                        va.va_size = bfp->l_start;
3323                        error = smbfssetattr(vp, &va, 0, cr);
3324                } else
3325                        error = EINVAL;
3326        }

3328        return (error);
3329 }

3331 /* ARGSUSED */
3332 static int
3333 smbfs_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
3334        caller_context_t *ct)
3335 {
3336        vfs_t *vfs;
3337        smbmntinfo_t *smi;
3338        struct smb_share *ssp;

3340        vfs = vp->v_vfsp;
3341        smi = VFTOSMI(vfs);

3343        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3344                return (EIO);

3346        if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3347                return (EIO);

3349        switch (cmd) {
3350        case _PC_FILESIZEBITS:
3351                ssp = smi->smi_share;
3352                if (SSTOVC(ssp)->vc_sopt.sv_caps & SMB_CAP_LARGE_FILES)
3353                        *valp = 64;
3354                else
3355                        *valp = 32;
3356                break;

3358        case _PC_LINK_MAX:
3359                /* We only ever report one link to an object */
3360                *valp = 1;
3361                break;

3363        case _PC_ACL_ENABLED:
3364                /*
3365                 * Always indicate that ACLs are enabled and
3366                 * that we support ACE_T format, otherwise
3367                 * libsec will ask for ACLENT_T format data
3368                 * which we don't support.
3369                 */
3370                *valp = _ACL_ACE_ENABLED;
3371                break;

3373        case _PC_SYMLINK_MAX:   /* No symlinks until we do Unix extensions */
3374                *valp = 0;
3375                break;

3377        case _PC_XATTR_EXISTS:
3378                if (vfs->vfs_flag & VFS_XATTR) {
3379                        *valp = smbfs_xa_exists(vp, cr);
```



```
3380                        break;
3381                }
3382                return (EINVAL);

3384        case _PC_TIMESTAMP_RESOLUTION:
3385                /*
3386                 * Windows times are tenths of microseconds
3387                 * (multiples of 100 nanoseconds).
3388                 */
3389                *valp = 100L;
3390                break;

3392        default:
3393                return (fs_pathconf(vp, cmd, valp, cr, ct));
3394        }
3395        return (0);
3396 }

3398 /* ARGSUSED */
3399 static int
3400 smbfs_getsecattr(vnode_t *vp, vsecattr_t *vsa, int flag, cred_t *cr,
3401        caller_context_t *ct)
3402 {
3403        vfs_t *vfsp;
3404        smbmntinfo_t *smi;
3405        int     error;
3406        uint_t  mask;

3408        vfsp = vp->v_vfsp;
3409        smi = VFTOSMI(vfsp);

3411        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3412                return (EIO);

3414        if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3415                return (EIO);

3417        /*
3418         * Our _pathconf indicates _ACL_ACE_ENABLED,
3419         * so we should only see VSA_ACE, etc here.
3420         * Note: vn_create asks for VSA_DFACLCNT,
3421         * and it expects ENOSYS and empty data.
3422         */
3423        mask = vsa->vsa_mask & (VSA_ACE | VSA_ACECNT |
3424            VSA_ACE_ACLFLAGS | VSA_ACE_ALLTYPES);
3425        if (mask == 0)
3426                return (ENOSYS);

3428        if (smi->smi_flags & SMI_ACL)
3429                error = smbfs_acl_getvsa(vp, vsa, flag, cr);
3430        else
3431                error = ENOSYS;

3433        if (error == ENOSYS)
3434                error = fs_fab_acl(vp, vsa, flag, cr, ct);

3436        return (error);
3437 }

3439 /* ARGSUSED */
3440 static int
3441 smbfs_setsecattr(vnode_t *vp, vsecattr_t *vsa, int flag, cred_t *cr,
3442        caller_context_t *ct)
3443 {
3444        vfs_t *vfsp;
3445        smbmntinfo_t *smi;
```

```
3446         int     error;
3447         uint_t  mask;
3448
3449         vfsp = vp->v_vfsp;
3450         smi = VFTOSMI(vfsp);
3451
3452         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3453                 return (EIO);
3454
3455         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3456                 return (EIO);
3457
3458         /*
3459          * Our _pathconf indicates _ACL_ACE_ENABLED,
3460          * so we should only see VSA_ACE, etc here.
3461          */
3462         mask = vsa->vsa_mask & (VSA_ACE | VSA_ACECNT);
3463         if (mask == 0)
3464                 return (ENOSYS);
3465
3466         if (vfsp->vfs_flag & VFS_RDONLY)
3467                 return (EROFS);
3468
3469         /*
3470          * Allow only the mount owner to do this.
3471          * See comments at smbfs_access_rwx.
3472          */
3473         error = secpolicy_vnode_setdac(cr, smi->smi_uid);
3474         if (error != 0)
3475                 return (error);
3476
3477         if (smi->smi_flags & SMI_ACL)
3478                 error = smbfs_acl_setvsa(vp, vsa, flag, cr);
3479         else
3480                 error = ENOSYS;
3481
3482         return (error);
3483 }


3486 /*
3487  * XXX
3488  * This op should eventually support PSARC 2007/268.
3489  */
3490 static int
3491 smbfs_shrlock(vnode_t *vp, int cmd, struct shrlock *shr, int flag, cred_t *cr,
3492         caller_context_t *ct)
3493 {
3494         if (curproc->p_zone != VTOSMI(vp)->smi_zone_ref.zref_zone)
3495                 return (EIO);
3496
3497         if (VTOSMI(vp)->smi_flags & SMI_LLOCK)
3498                 return (fs_shrlock(vp, cmd, shr, flag, cr, ct));
3499         else
3500                 return (ENOSYS);
3501 }

3503 /* correspond to bp_mapin() in bp_map.c */
3504 static int
3505 uio_page_mapin(uio_t * uiop, page_t * pp)
3506 {
3507         u_offset_t      off;
3508         size_t          size;
3509         pgcnt_t         npages;
3510         caddr_t         kaddr;
3511         pfn_t           pfnum;
```



```
3513         off = (uintptr_t) uiop->uio_loffset & PAGEOFFSET;
3514         size = P2ROUNDUP(uiop->uio_resid + off, PAGESIZE);
3515         npages = btop(size);
3516
3517         ASSERT(pp != NULL);
3518
3519         if (npages == 1 && kpm_enable) {
3520                 kaddr = hat_kpm_mapin(pp, NULL);
3521                 if (kaddr == NULL)
3522                         return (EFAULT);
3523
3524                 uiop->uio_iov->iov_base = kaddr + off;
3525                 uiop->uio_iov->iov_len = PAGESIZE - off;
3526
3527         } else {
3528                 kaddr = vmem_xalloc(heap_arena, size, PAGESIZE, 0, 0, NULL, NULL
3529                 if (kaddr == NULL)
3530                         return (EFAULT);
3531
3532                 uiop->uio_iov->iov_base = kaddr + off;
3533                 uiop->uio_iov->iov_len = size - off;
3534
3535                 /* map pages into kaddr */
3536                 uint_t          attr = PROT_READ | PROT_WRITE | HAT_NOSYNC;
3537                 while (npages-- > 0) {
3538                         pfnum = pp->p_pagenum;
3539                         pp = pp->p_next;
3540
3541                         hat_devload(kas.a_hat, kaddr, PAGESIZE, pfnum, attr, HAT
3542                         kaddr += PAGESIZE;
3543                 }
3544         }
3545         return (0);
3546 }

3548 /* correspond to bp_mapout() in bp_map.c */
3549 static void
3550 uio_page_mapout(uio_t * uiop, page_t * pp)
3551 {
3552         u_offset_t      off;
3553         size_t          size;
3554         pgcnt_t         npages;
3555         caddr_t         kaddr;
3556
3557         kaddr = uiop->uio_iov->iov_base;
3558         off = (uintptr_t) kaddr & PAGEOFFSET;
3559         size = P2ROUNDUP(uiop->uio_iov->iov_len + off, PAGESIZE);
3560         npages = btop(size);
3561
3562         ASSERT(pp != NULL);
3563
3564         kaddr = (caddr_t) ((uintptr_t) kaddr & MMU_PAGEMASK);
3565
3566         if (npages == 1 && kpm_enable) {
3567                 hat_kpm_mapout(pp, NULL, kaddr);
3568
3569         } else {
3570                 hat_unload(kas.a_hat, (void *) kaddr, size,
3571                         HAT_UNLOAD_NOSYNC | HAT_UNLOAD_UNLOCK);
3572                 vmem_free(heap_arena, (void *) kaddr, size);
3573         }
3574         uiop->uio_iov->iov_base = 0;
3575         uiop->uio_iov->iov_len = 0;
3576 }
```

```
3578 static int
3579 smbfs_map(vnode_t * vp, offset_t off, struct as * as, caddr_t * addrp,
3580        size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t * cr,
3581           caller_context_t * ct)
3582 {
3583         smbnode_t      *np;
3584         smbmntinfo_t   *smi;
3585         struct vattr    va;
3586         segvn_crargs_t  vn_a;
3587         int             error;

3589         np = VTOSMB(vp);
3590         smi = VTOSMI(vp);

3592         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3593                 return (EIO);

3595         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3596                 return (EIO);

3598         if (vp->v_flag & VNOMAP || vp->v_flag & VNOCACHE)
3599                 return (EAGAIN);

3601         if (vp->v_type != VREG)
3602                 return (ENODEV);

3604         va.va_mask = AT_ALL;
3605         if (error = smbfsgetattr(vp, &va, cr))
3606                 return (error);

3608         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
3609                 return (EINTR);

3611         if (MANDLOCK(vp, va.va_mode)) {
3612                 error = EAGAIN;
3613                 goto out;
3614         }
3615         as_rangelock(as);
3616         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);

3618         if (error != 0) {
3619                 as_rangeunlock(as);
3620                 goto out;
3621         }
3622         vn_a.vp = vp;
3623         vn_a.offset = off;
3624         vn_a.type = flags & MAP_TYPE;
3625         vn_a.prot = prot;
3626         vn_a.maxprot = maxprot;
3627         vn_a.flags = flags & ~MAP_TYPE;
3628         vn_a.cred = cr;
3629         vn_a.amp = NULL;
3630         vn_a.szc = 0;
3631         vn_a.lgrp_mem_policy_flags = 0;

3633         error = as_map(as, *addrp, len, segvn_create, &vn_a);

3635         as_rangeunlock(as);

3637 out:
3638         smbfs_rw_exit(&np->r_lkserlock);

3640         return (error);
3641 }

3643 static int
```



```
3644 smbfs_addmap(vnode_t * vp, offset_t off, struct as * as, caddr_t addr,
3645        size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t * cr,
3646             caller_context_t * ct)
3647 {
3648         atomic_add_long((ulong_t *) & VTOSMB(vp)->r_mapcnt, btopr(len));
3649         return (0);
3650 }

3652 static int
3653 smbfs_delmap(vnode_t * vp, offset_t off, struct as * as, caddr_t addr,
3654          size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t * cr,
3655              caller_context_t * ct)
3656 {

3658         smbnode_t      *np;

3660         atomic_add_long((ulong_t *) & VTOSMB(vp)->r_mapcnt, -btopr(len));

3662         /*
3663          * mark RDIRTY here, will be used to check if a file is dirty when
3664          * unmount smbfs
3665          */
3666         if (vn_has_cached_data(vp) && !vn_is_readonly(vp) && (maxprot & PROT_WRI
3667             && (flags == MAP_SHARED)) {
3668                 np = VTOSMB(vp);
3669                 mutex_enter(&np->r_statelock);
3670                 np->r_flags |= RDIRTY;
3671                 mutex_exit(&np->r_statelock);
3672         }
3673         return (0);
3674 }

3676 static int
3677 smbfs_putpage(vnode_t * vp, offset_t off, size_t len, int flags,
3678               cred_t * cr, caller_context_t * ct)
3679 {

3681         smbnode_t      *np;
3682         size_t          io_len;
3683         u_offset_t      io_off;
3684         u_offset_t      eoff;
3685         int             error = 0;
3686         page_t         *pp;
3687         int             rdirty;

3689         np = VTOSMB(vp);

3691         if (len == 0) {

3693                 /* will flush the whole file, so clear RDIRTY */
3694                 if (off == (u_offset_t) 0 && (np->r_flags & RDIRTY)) {
3695                         mutex_enter(&np->r_statelock);
3696                         rdirty = np->r_flags & RDIRTY;
3697                         np->r_flags &= ~RDIRTY;
3698                         mutex_exit(&np->r_statelock);
3699                 } else
3700                         rdirty = 0;

3702                 error = pvn_vplist_dirty(vp, off, smbfs_putapage, flags, cr);

3704                 /*
3705                  * if failed and the vnode was dirty before and we aren't
3706                  * forcibly invalidating pages, then mark RDIRTY again.
3707                  */
3708                 if (error && rdirty &&
3709                     (flags & (B_INVAL | B_FORCE)) != (B_INVAL | B_FORCE)) {
```

```
3710                        mutex_enter(&np->r_statelock);
3711                        np->r_flags |= RDIRTY;
3712                        mutex_exit(&np->r_statelock);
3713                }
3714        } else {

3716                eoff = off + len;

3718                mutex_enter(&np->r_statelock);
3719                if (eoff > np->r_size)
3720                        eoff = np->r_size;
3721                mutex_exit(&np->r_statelock);

3723                for (io_off = off; io_off < eoff; io_off += io_len) {
3724                        if ((flags & B_INVAL) || (flags & B_ASYNC) == 0) {
3725                                pp = page_lookup(vp, io_off,
3726                                                 (flags & (B_INVAL | B_FREE) ? S
3727                        } else {
3728                                pp = page_lookup_nowait(vp, io_off,
3729                                    (flags & B_FREE) ? SE_EXCL : SE_SHARED);
3730                        }

3732                        if (pp == NULL || !pvn_getdirty(pp, flags))
3733                                io_len = PAGESIZE;
3734                        else {
3735                                error = smbfs_putapage(vp, pp, &io_off, &io_len,
3736                        }
3737                }

3739        }

3741        return (error);
3742 }

3744 static int
3745 smbfs_putapage(vnode_t * vp, page_t * pp, u_offset_t * offp, size_t * lenp,
3746                int flags, cred_t * cr)
3747 {

3749        struct smb_cred scred;
3750        smbnode_t      *np;
3751        smbmntinfo_t   *smi;
3752        smb_share_t    *ssp;
3753        uio_t           uio;
3754        iovec_t         uiov, uiov_bak;

3756        size_t          io_len;
3757        u_offset_t      io_off;
3758        size_t          limit;
3759        size_t          bsize;
3760        size_t          blksize;
3761        u_offset_t      blkoff;
3762        int             error;

3764        np = VTOSMB(vp);
3765        smi = VTOSMI(vp);
3766        ssp = smi->smi_share;

3768        /* do block io, get a kluster of dirty pages in a block. */
3769        bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
3770        blkoff = pp->p_offset / bsize;
3771        blkoff *= bsize;
3772        blksize = roundup(bsize, PAGESIZE);

3774        pp = pvn_write_kluster(vp, pp, &io_off, &io_len, blkoff, blksize, flags)
```



```
3776        ASSERT(pp->p_offset >= blkoff);

3778        if (io_off + io_len > blkoff + blksize) {
3779                ASSERT((io_off + io_len) - (blkoff + blksize) < PAGESIZE);
3780        }

3782        /* Don't allow put pages beyond EOF */
3783        mutex_enter(&np->r_statelock);
3784        limit=MIN(np->r_size, blkoff + blksize);
3785        mutex_exit(&np->r_statelock);

3787        if (io_off >= limit) {
3788                error = 0;
3789                goto out;
3790        } else if (io_off + io_len > limit) {
3791                int             npages = btopr(limit - io_off);
3792                page_t         *trunc;
3793                page_list_break(&pp, &trunc, npages);
3794                if (trunc)
3795                        pvn_write_done(trunc, flags);
3796                io_len = limit - io_off;
3797        }

3799        /*
3800         * Taken from NFS4. The RMODINPROGRESS flag makes sure that
3801         * smbfs_putapage() sees a consistent value of r_size. RMODINPROGRESS
3802         * is set in writenp(). When RMODINPROGRESS is set it indicates that
3803         * a uiomove() is in progress and the r_size has not been made
3804         * consistent with the new size of the file. When the uiomove()
3805         * completes the r_size is updated and the RMODINPROGRESS flag is
3806         * cleared.
3807         *
3808         * The RMODINPROGRESS flag makes sure that smbfs_putapage() sees a
3809         * consistent value of r_size. Without this handshaking, it is
3810         * possible that smbfs_putapage() picks  up the old value of r_size
3811         * before the uiomove() in writenp() completes. This will result in
3812         * the write through smbfs_putapage() being dropped.
3813         *
3814         * More precisely, there is a window between the time the uiomove()
3815         * completes and the time the r_size is updated. If a VOP_PUTPAGE()
3816         * operation intervenes in this window, the page will be picked up,
3817         * because it is dirty (it will be unlocked, unless it was
3818         * pagecreate'd). When the page is picked up as dirty, the dirty bit
3819         * is reset (pvn_getdirty()). In smbfs_putapage(), r_size is checked.
3820         * This will still be the old size. Therefore the page will not be
3821         * written out. When segmap_release() calls VOP_PUTPAGE(), the page
3822         * will be found to be clean and the write will be dropped.
3823         */
3824        if (np->r_flags & RMODINPROGRESS) {

3826                mutex_enter(&np->r_statelock);
3827                if ((np->r_flags & RMODINPROGRESS) &&
3828                    np->r_modaddr + MAXBSIZE > io_off &&
3829                    np->r_modaddr < io_off + io_len) {
3830                        page_t         *plist;
3831                        /*
3832                         * A write is in progress for this region of the
3833                         * file. If we did not detect RMODINPROGRESS here,
3834                         * the data beyond the file size won't be write out.
3835                         * We end up losing data. So we decide to set the
3836                         * modified bit on each page in the page list and
3837                         * mark the smbnode with RDIRTY. This write will be
3838                         * restarted at some later time.
3839                         */
3840                        plist = pp;
3841                        while (plist != NULL) {
```

```
3842                                pp = plist;
3843                                page_sub(&plist, pp);
3844                                hat_setmod(pp);
3845                                page_io_unlock(pp);
3846                                page_unlock(pp);
3847                        }
3848                        np->r_flags |= RDIRTY;
3849                        mutex_exit(&np->r_statelock);
3850                        if (offp)
3851                                *offp = io_off;
3852                        if (lenp)
3853                                *lenp = io_len;
3854                        return (0);
3855                }
3856                mutex_exit(&np->r_statelock);
3857        }

3859        if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
3860                return (EINTR);
3861        smb_credinit(&scred, cr);

3863        if (np->n_vcgenid != ssp->ss_vcgenid)
3864                error = ESTALE;
3865        else {
3866                /* just use uio instead of buf, since smb_rwuio need uio. */
3867                uiov.iov_base = 0;
3868                uiov.iov_len = 0;
3869                uio.uio_iov = &uiov;
3870                uio.uio_iovcnt = 1;
3871                uio.uio_loffset = io_off;
3872                uio.uio_resid = io_len;
3873                uio.uio_segflg = UIO_SYSSPACE;
3874                uio.uio_llimit = MAXOFFSET_T;
3875                /* map pages into kernel address space, and setup uio. */
3876                error = uio_page_mapin(&uio, pp);
3877                if (error == 0) {
3878                        uiov_bak.iov_base = uiov.iov_base;
3879                        uiov_bak.iov_len = uiov.iov_len;
3880                        error = smb_rwuio(ssp, np->n_fid, UIO_WRITE, &uio, &scre
3881                        if (error == 0) {
3882                                mutex_enter(&np->r_statelock);
3883                                np->n_flag |= (NFLUSHWIRE | NATTRCHANGED);
3884                                mutex_exit(&np->r_statelock);
3885                                (void) smbfs_smb_flush(np, &scred);
3886                        }
3887                        /* unmap pages from kernel address space. */
3888                        uio.uio_iov = &uiov_bak;
3889                        uio_page_mapout(&uio, pp);
3890                }
3891        }

3893        smb_credrele(&scred);
3894        smbfs_rw_exit(&np->r_lkserlock);

3896 out:
3897        pvn_write_done(pp, ((error) ? B_ERROR : 0) | B_WRITE | flags);

3899        if (offp)
3900                *offp = io_off;
3901        if (lenp)
3902                *lenp = io_len;

3904        return (error);
3905 }

3907 static int
```



```
3908 smbfs_getpage(vnode_t * vp, offset_t off, size_t len, uint_t * protp,
3909               page_t * pl[], size_t plsz, struct seg * seg, caddr_t addr,
3910               enum seg_rw rw, cred_t * cr, caller_context_t * ct)
3911 {

3913        smbnode_t      *np;
3914        int             error;

3916        /* these pages have all protections. */
3917        if (protp)
3918                *protp = PROT_ALL;

3920        np = VTOSMB(vp);

3922        /* Don't allow get pages beyond EOF, unless it's segkmap. */
3923        mutex_enter(&np->r_statelock);
3924        if (off + len > np->r_size + PAGESIZE && seg != segkmap){
3925                mutex_exit(&np->r_statelock);
3926                return (EFAULT);
3927        }
3928        mutex_exit(&np->r_statelock);

3930        if (len <= PAGESIZE) {
3931                error = smbfs_getapage(vp, off, len, protp, pl, plsz, seg, addr,
3932                                       cr);
3933        } else {
3934                error = pvn_getpages(smbfs_getapage, vp, off, len, protp, pl, pl
3935                                     addr, rw, cr);
3936        }

3938        return (error);
3939 }

3941 static int
3942 smbfs_getapage(vnode_t * vp, u_offset_t off, size_t len,
3943  uint_t * protp, page_t * pl[], size_t plsz, struct seg * seg, caddr_t addr,
3944                enum seg_rw rw, cred_t * cr)
3945 {

3947        smbnode_t      *np;
3948        smbmntinfo_t   *smi;
3949        smb_share_t    *ssp;
3950        smb_cred_t      scred;

3952        page_t         *pp;
3953        uio_t           uio;
3954        iovec_t         uiov, uiov_bak;

3956        u_offset_t      blkoff;
3957        size_t          bsize;
3958        size_t          blksize;

3960        u_offset_t      io_off;
3961        size_t          io_len;
3962        size_t          pages_len;

3964        int             error = 0;

3966        np = VTOSMB(vp);
3967        smi = VTOSMI(vp);
3968        ssp = smi->smi_share;

3970        /* if pl is null,it's meaningless */
3971        if (pl == NULL)
3972                return (EFAULT);
```

```
again:
        if (page_exists(vp, off) == NULL) {
                if (rw == S_CREATE) {
                        /* just return a empty page if asked to create. */
                        if ((pp = page_create_va(vp, off, PAGESIZE, PG_WAIT | PG
                                goto again;
                        pages_len = PAGESIZE;
                } else {

                        /*
                         * do block io, get a kluster of non-exist pages in a
                         * block.
                         */
                        bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
                        blkoff = off / bsize;
                        blkoff *= bsize;
                        blksize = roundup(bsize, PAGESIZE);

                        pp = pvn_read_kluster(vp, off, seg, addr, &io_off, &io_l

                        if (pp == NULL)
                                goto again;

                        pages_len = io_len;

                        /* Don't need to get pages from server if it's segkmap
                         * that reads beyond EOF. */
                        mutex_enter(&np->r_statelock);
                        if (io_off >= np->r_size && seg == segkmap) {
                                mutex_exit(&np->r_statelock);
                                error = 0;
                                goto out;
                        } else if (io_off + io_len > np->r_size) {
                                int             npages = btopr(np->r_size - io_o
                                page_t         *trunc;

                                page_list_break(&pp, &trunc, npages);
                                if (trunc)
                                        pvn_read_done(trunc, 0);
                                io_len = np->r_size - io_off;
                        }
                        mutex_exit(&np->r_statelock);

                        if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBI
                                return EINTR;
                        smb_credinit(&scred, cr);

                        /*
                         * just use uio instead of buf, since smb_rwuio need
                         * uio.
                         */
                        uiov.iov_base = 0;
                        uiov.iov_len = 0;
                        uio.uio_iov = &uiov;
                        uio.uio_iovcnt = 1;
                        uio.uio_loffset = io_off;
                        uio.uio_resid = io_len;
                        uio.uio_segflg = UIO_SYSSPACE;
                        uio.uio_llimit = MAXOFFSET_T;

                        /*
                         * map pages into kernel address space, and setup
                         * uio.
                         */
                        error = uio_page_mapin(&uio, pp);
                        if (error == 0) {
```



```
                                uiov_bak.iov_base = uiov.iov_base;
                                uiov_bak.iov_len = uiov.iov_len;
                                error = smb_rwuio(ssp, np->n_fid, UIO_READ, &uio
                                /* unmap pages from kernel address space. */
                                uio.uio_iov = &uiov_bak;
                                uio_page_mapout(&uio, pp);
                        }
                        smb_credrele(&scred);
                        smbfs_rw_exit(&np->r_lkserlock);
                }
        } else {
                se_t            se = rw == S_CREATE ? SE_EXCL : SE_SHARED;
                if ((pp = page_lookup(vp, off, se)) == NULL) {
                        goto again;
                }
        }

out:
        if (pp) {
                if (error) {
                        pvn_read_done(pp, B_ERROR);
                } else {
                        /* init page list, unlock pages. */
                        pvn_plist_init(pp, pl, plsz, off, pages_len, rw);
                }
        }
        return (error);
}

/* correspond to nfs_invalidate_pages() in nfs_client.c */
void
smbfs_invalidate_pages(vnode_t * vp, u_offset_t off, cred_t * cr)
{

        smbnode_t      *np;

        np = VTOSMB(vp);
        /* will flush the whole file, so clear RDIRTY */
        if (off == (u_offset_t) 0 && (np->r_flags & RDIRTY)) {
                mutex_enter(&np->r_statelock);
                np->r_flags &= ~RDIRTY;
                mutex_exit(&np->r_statelock);
        }
        (void) pvn_vplist_dirty(vp, off, smbfs_putapage, B_INVAL | B_TRUNC, cr);
}
#endif /* ! codereview */
```