```
**********************************************************
   10945 Fri Jul 20 12:37:50 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_ */
```

```
**********************************************************
   29723 Fri Jul 20 12:37:51 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  * Flush all vnodes in this (or every) vfs.
1047  * Used by nfs_sync and by nfs_unmount.
1048  */
1049 /*ARGSUSED*/
1050 void
```

```
1051 smbfs_rflush(struct vfs *vfsp, cred_t *cr) {

1053     smbmntinfo_t *mi;
1054     smbnode_t *np;
1055     vnode_t *vp;

1057     long num, cnt;

1059     vnode_t **vplist;

1061     mi = VFTOSMI(vfsp);

1063     cnt = 0;
1064     num = mi->smi_hash_avl.avl_numnodes;
1065     vplist = kmem_alloc(num * sizeof (vnode_t*), KM_SLEEP);

1067     rw_enter(&mi->smi_hash_lk, RW_READER);
1068     for (np = avl_first(&mi->smi_hash_avl); np != NULL;
1069             np = avl_walk(&mi->smi_hash_avl, np, AVL_AFTER)) {
1070         vp = SMBTOV(np);
1071         if (vn_is_readonly(vp))
1072             continue;

1074         if (vn_has_cached_data(vp) && (np->r_flags & RDIRTY || np->r_mapcnt > 0)
1075             VN_HOLD(vp);
1076             vplist[cnt++] = vp;
1077             if (cnt == num)
1078                 break;
1079         }
1080     }
1081     rw_exit(&mi->smi_hash_lk);

1083     while (cnt-- > 0) {
1084         vp = vplist[cnt];
1085         (void) VOP_PUTPAGE(vp, 0, 0, 0, cr, NULL);
1086         VN_RELE(vp);
1087     }

1089     kmem_free(vplist, num * sizeof (vnode_t*));
 180 smbfs_rflush(struct vfs *vfsp, cred_t *cr)
 181 {
 182         /* Todo: mmap support. */
1090 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   93502 Fri Jul 20 12:37:51 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);



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                 smbfs_rele_fid(np, &scred);
548         } else if (np->r_mapcnt == 0) {
549                 /*
550                  * Before otW close, make sure dirty pages written back.
551                  */
552                 if ((flag & FWRITE) && vn_has_cached_data(vp)) {
553                         /* smbfs_putapage() will acquire shared lock, so release
554                          * exclusive lock temporally.
555                          */
556                         smbfs_rw_exit(&np->r_lkserlock);

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

560                         /* acquire exclusive lock again. */
561                         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0
562                 }
563 #endif /* ! codereview */
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)
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         np = VTOSMB(vp);
668         smi = VTOSMI(vp);
669         ssp = smi->smi_share;
670
671         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
672                 return (EIO);
673
674         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
675                 return (EIO);
676
677         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_READER));
678
679         if (vp->v_type != VREG)
680                 return (EISDIR);
681
682         if (uiop->uio_resid == 0)
683                 return (0);
684
685         /*
686          * Like NFS3, just check for 63-bit overflow.
687          * Our SMB layer takes care to return EFBIG
688          * when it has to fallback to a 32-bit call.
689          */
690         endoff = uiop->uio_loffset + uiop->uio_resid;
691         if (uiop->uio_loffset < 0 || endoff < 0)
692                 return (EINVAL);
693
694         /* get vnode attributes from server */
695         va.va_mask = AT_SIZE | AT_MTIME;
696         if (error = smbfsgetattr(vp, &va, cr))
697                 return (error);
698
699         /* Update mtime with mtime from server here? */
```

```
701         /* if offset is beyond EOF, read nothing */
702         if (uiop->uio_loffset >= va.va_size)
703                 return (0);

705         /*
706          * Limit the read to the remaining file size.
707          * Do this by temporarily reducing uio_resid
708          * by the amount the lies beyoned the EOF.
709          */
710         if (endoff > va.va_size) {
711                 past_eof = (ssize_t)(endoff - va.va_size);
712                 uiop->uio_resid -= past_eof;
713         } else
714                 past_eof = 0;

716         /* Shared lock for n_fid use in smb_rwuio */
717         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
718                 return (EINTR);
719         smb_credinit(&scred, cr);

721         /* After reconnect, n_fid is invalid */
722         if (np->n_vcgenid != ssp->ss_vcgenid)
723                 error = ESTALE;
724         else
725                 error = smb_rwuio(ssp, np->n_fid, UIO_READ,
726                     uiop, &scred, smb_timo_read);

728         smb_credrele(&scred);
729         smbfs_rw_exit(&np->r_lkserlock);

731         /* undo adjustment of resid */
732         uiop->uio_resid += past_eof;

734         return (error);
735 }


738 /* ARGSUSED */
739 static int
740 smbfs_write(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
741         caller_context_t *ct)
742 {
743         struct smb_cred scred;
744         struct vattr    va;
745         smbnode_t       *np;
746         smbmntinfo_t    *smi;
747         smb_share_t     *ssp;
748         offset_t        endoff, limit;
749         ssize_t         past_limit;
750         int             error, timo;

752         np = VTOSMB(vp);
753         smi = VTOSMI(vp);
754         ssp = smi->smi_share;

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

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

762         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_WRITER));

764         if (vp->v_type != VREG)
765                 return (EISDIR);
```



```
767         if (uiop->uio_resid == 0)
768                 return (0);

770         /*
771          * Handle ioflag bits: (FAPPEND|FSYNC|FDSYNC)
772          */
773         if (ioflag & (FAPPEND | FSYNC)) {
774                 if (np->n_flag & NMODIFIED) {
775                         smbfs_attrcache_remove(np);
776                         /* XXX: smbfs_vinvalbuf? */
777                 }
778         }
779         if (ioflag & FAPPEND) {
780                 /*
781                  * File size can be changed by another client
782                  */
783                 va.va_mask = AT_SIZE;
784                 if (error = smbfsgetattr(vp, &va, cr))
785                         return (error);
786                 uiop->uio_loffset = va.va_size;
787         }

789         /*
790          * Like NFS3, just check for 63-bit overflow.
791          */
792         endoff = uiop->uio_loffset + uiop->uio_resid;
793         if (uiop->uio_loffset < 0 || endoff < 0)
794                 return (EINVAL);

796         /*
797          * Check to make sure that the process will not exceed
798          * its limit on file size.  It is okay to write up to
799          * the limit, but not beyond.  Thus, the write which
800          * reaches the limit will be short and the next write
801          * will return an error.
802          *
803          * So if we're starting at or beyond the limit, EFBIG.
804          * Otherwise, temporarily reduce resid to the amount
805          * the falls after the limit.
806          */
807         limit = uiop->uio_llimit;
808         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
809                 limit = MAXOFFSET_T;
810         if (uiop->uio_loffset >= limit)
811                 return (EFBIG);
812         if (endoff > limit) {
813                 past_limit = (ssize_t)(endoff - limit);
814                 uiop->uio_resid -= past_limit;
815         } else
816                 past_limit = 0;

818         /* Timeout: longer for append. */
819         timo = smb_timo_write;
820         if (endoff > np->r_size)
821                 timo = smb_timo_append;

823         /* Shared lock for n_fid use in smb_rwuio */
824         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
825                 return (EINTR);
826         smb_credinit(&scred, cr);

828         /* After reconnect, n_fid is invalid */
829         if (np->n_vcgenid != ssp->ss_vcgenid)
830                 error = ESTALE;
831         else
832                 error = smb_rwuio(ssp, np->n_fid, UIO_WRITE,
```

```
833                     uiop, &scred, timo);

835         if (error == 0) {
836                 mutex_enter(&np->r_statelock);
837                 np->n_flag |= (NFLUSHWIRE | NATTRCHANGED);
838                 if (uiop->uio_loffset > (offset_t)np->r_size)
839                         np->r_size = (len_t)uiop->uio_loffset;
840                 mutex_exit(&np->r_statelock);
841                 if (ioflag & (FSYNC|FDSYNC)) {
842                         /* Don't error the I/O if this fails. */
843                         (void) smbfs_smb_flush(np, &scred);
844                 }
845         }

847         smb_credrele(&scred);
848         smbfs_rw_exit(&np->r_lkserlock);

850         /* undo adjustment of resid */
851         uiop->uio_resid += past_limit;

853         return (error);
854 }


857 /* ARGSUSED */
858 static int
859 smbfs_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag,
860         cred_t *cr, int *rvalp, caller_context_t *ct)
861 {
862         int             error;
863         smbmntinfo_t    *smi;

865         smi = VTOSMI(vp);

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

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

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

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

880                 /*
881                  * The following two ioctls are used by bfu.
882                  * Silently ignore to avoid bfu errors.
883                  */
884         case _FIOGDIO:
885         case _FIOSDIO:
886                 error = 0;
887                 break;

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

894                 /*
895                  * Allow get/set with "raw" security descriptor (SD) data.
896                  * Useful for testing, diagnosing idmap problems, etc.
897                  */
898         case SMBFSIO_GETSD:
```



```
899                 error = smbfs_acl_iocget(vp, arg, flag, cr);
900                 break;

902         case SMBFSIO_SETSD:
903                 error = smbfs_acl_iocset(vp, arg, flag, cr);
904                 break;

906         default:
907                 error = ENOTTY;
908                 break;
909         }

911         return (error);
912 }


915 /*
916  * Return either cached or remote attributes. If get remote attr
917  * use them to check and invalidate caches, then cache the new attributes.
918  *
919  * XXX
920  * This op should eventually support PSARC 2007/315, Extensible Attribute
921  * Interfaces, for richer metadata.
922  */
923 /* ARGSUSED */
924 static int
925 smbfs_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
926         caller_context_t *ct)
927 {
928         smbnode_t *np;
929         smbmntinfo_t *smi;

931         smi = VTOSMI(vp);

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

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

939         /*
940          * If it has been specified that the return value will
941          * just be used as a hint, and we are only being asked
942          * for size, fsid or rdevid, then return the client's
943          * notion of these values without checking to make sure
944          * that the attribute cache is up to date.
945          * The whole point is to avoid an over the wire GETATTR
946          * call.
947          */
948         np = VTOSMB(vp);
949         if (flags & ATTR_HINT) {
950                 if (vap->va_mask ==
951                     (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
952                         mutex_enter(&np->r_statelock);
953                         if (vap->va_mask | AT_SIZE)
954                                 vap->va_size = np->r_size;
955                         if (vap->va_mask | AT_FSID)
956                                 vap->va_fsid = vp->v_vfsp->vfs_dev;
957                         if (vap->va_mask | AT_RDEV)
958                                 vap->va_rdev = vp->v_rdev;
959                         mutex_exit(&np->r_statelock);
960                         return (0);
961                 }
962         }

964         return (smbfsgetattr(vp, vap, cr));
```

```
 965 }
 967 /* smbfsgetattr() in smbfs_client.c */
 969 /*
 970  * XXX
 971  * This op should eventually support PSARC 2007/315, Extensible Attribute
 972  * Interfaces, for richer metadata.
 973  */
 974 /*ARGSUSED4*/
 975 static int
 976 smbfs_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
 977                 caller_context_t *ct)
 978 {
 979         vfs_t           *vfsp;
 980         smbmntinfo_t    *smi;
 981         int             error;
 982         uint_t          mask;
 983         struct vattr    oldva;
 985         vfsp = vp->v_vfsp;
 986         smi = VFTOSMI(vfsp);
 988         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
 989                 return (EIO);
 991         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
 992                 return (EIO);
 994         mask = vap->va_mask;
 995         if (mask & AT_NOSET)
 996                 return (EINVAL);
 998         if (vfsp->vfs_flag & VFS_RDONLY)
 999                 return (EROFS);
1001         /*
1002          * This is a _local_ access check so that only the owner of
1003          * this mount can set attributes.  With ACLs enabled, the
1004          * file owner can be different from the mount owner, and we
1005          * need to check the _mount_ owner here.  See _access_rwx
1006          */
1007         bzero(&oldva, sizeof (oldva));
1008         oldva.va_mask = AT_TYPE | AT_MODE;
1009         error = smbfsgetattr(vp, &oldva, cr);
1010         if (error)
1011                 return (error);
1012         oldva.va_mask |= AT_UID | AT_GID;
1013         oldva.va_uid = smi->smi_uid;
1014         oldva.va_gid = smi->smi_gid;
1016         error = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
1017             smbfs_accessx, vp);
1018         if (error)
1019                 return (error);
1021         if (mask & (AT_UID | AT_GID)) {
1022                 if (smi->smi_flags & SMI_ACL)
1023                         error = smbfs_acl_setids(vp, vap, cr);
1024                 else
1025                         error = ENOSYS;
1026                 if (error != 0) {
1027                         SMBVDEBUG("error %d seting UID/GID on %s",
1028                             error, VTOSMB(vp)->n_rpath);
1029                         /*
1030                          * It might be more correct to return the
```



```
1031                          * error here, but that causes complaints
1032                          * when root extracts a cpio archive, etc.
1033                          * So ignore this error, and go ahead with
1034                          * the rest of the setattr work.
1035                          */
1036                 }
1037         }
1039         return (smbfssetattr(vp, vap, flags, cr));
1040 }
1042 /*
1043  * Mostly from Darwin smbfs_setattr()
1044  * but then modified a lot.
1045  */
1046 /* ARGSUSED */
1047 static int
1048 smbfssetattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr)
1049 {
1050         int             error = 0;
1051         smbnode_t       *np = VTOSMB(vp);
1052         uint_t          mask = vap->va_mask;
1053         struct timespec *mtime, *atime;
1054         struct smb_cred scred;
1055         int             cerror, modified = 0;
1056         unsigned short  fid;
1057         int have_fid = 0;
1058         uint32_t rights = 0;
1060         ASSERT(curproc->p_zone == VTOSMI(vp)->smi_zone_ref.zref_zone);
1062         /*
1063          * There are no settable attributes on the XATTR dir,
1064          * so just silently ignore these.  On XATTR files,
1065          * you can set the size but nothing else.
1066          */
1067         if (vp->v_flag & V_XATTRDIR)
1068                 return (0);
1069         if (np->n_flag & N_XATTR) {
1070                 if (mask & AT_TIMES)
1071                         SMBVDEBUG("ignore set time on xattr\n");
1072                 mask &= AT_SIZE;
1073         }
1075         /*
1076          * If our caller is trying to set multiple attributes, they
1077          * can make no assumption about what order they are done in.
1078          * Here we try to do them in order of decreasing likelihood
1079          * of failure, just to minimize the chance we'll wind up
1080          * with a partially complete request.
1081          */
1083         /* Shared lock for (possible) n_fid use. */
1084         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
1085                 return (EINTR);
1086         smb_credinit(&scred, cr);
1088         /*
1089          * Will we need an open handle for this setattr?
1090          * If so, what rights will we need?
1091          */
1092         if (mask & (AT_ATIME | AT_MTIME)) {
1093                 rights |=
1094                     SA_RIGHT_FILE_WRITE_ATTRIBUTES;
1095         }
1096         if (mask & AT_SIZE) {
```

```
1097                    rights |=
1098                        SA_RIGHT_FILE_WRITE_DATA |
1099                        SA_RIGHT_FILE_APPEND_DATA;
1100            }
1101
1102            /*
1103             * Only SIZE really requires a handle, but it's
1104             * simpler and more reliable to set via a handle.
1105             * Some servers like NT4 won't set times by path.
1106             * Also, we're usually setting everything anyway.
1107             */
1108            if (mask & (AT_SIZE | AT_ATIME | AT_MTIME)) {
1109                    error = smbfs_smb_tmpopen(np, rights, &scred, &fid);
1110                    if (error) {
1111                            SMBVDEBUG("error %d opening %s\n",
1112                                error, np->n_rpath);
1113                            goto out;
1114                    }
1115                    have_fid = 1;
1116            }
1117
1118            /*
1119             * If the server supports the UNIX extensions, right here is where
1120             * we'd support changes to uid, gid, mode, and possibly va_flags.
1121             * For now we claim to have made any such changes.
1122             */
1123
1124            if (mask & AT_SIZE) {
1125                    /*
1126                     * If the new file size is less than what the client sees as
1127                     * the file size, then just change the size and invalidate
1128                     * the pages.
1129                     * I am commenting this code at present because the function
1130                     * smbfs_putapage() is not yet implemented.
1131                     */
1132
1133                    /*
1134                     * Set the file size to vap->va_size.
1135                     */
1136                    ASSERT(have_fid);
1137                    error = smbfs_smb_setfsize(np, fid, vap->va_size, &scred);
1138                    if (error) {
1139                            SMBVDEBUG("setsize error %d file %s\n",
1140                                error, np->n_rpath);
1141                    } else {
1142                            /*
1143                             * Darwin had code here to zero-extend.
1144                             * Tests indicate the server will zero-fill,
1145                             * so looks like we don't need to do this.
1146                             * Good thing, as this could take forever.
1147                             *
1148                             * XXX: Reportedly, writing one byte of zero
1149                             * at the end offset avoids problems here.
1150                             */
1151                            mutex_enter(&np->r_statelock);
1152                            np->r_size = vap->va_size;
1153                            mutex_exit(&np->r_statelock);
1154                            modified = 1;
1155                    }
1156            }
1157
1158            /*
1159             * XXX: When Solaris has create_time, set that too.
1160             * Note: create_time is different from ctime.
1161             */
1162            mtime = ((mask & AT_MTIME) ? &vap->va_mtime : 0);
```



```
1163            atime = ((mask & AT_ATIME) ? &vap->va_atime : 0);
1164
1165            if (mtime || atime) {
1166                    /*
1167                     * Always use the handle-based set attr call now.
1168                     * Not trying to set DOS attributes here so pass zero.
1169                     */
1170                    ASSERT(have_fid);
1171                    error = smbfs_smb_setfattr(np, fid,
1172                        0, mtime, atime, &scred);
1173                    if (error) {
1174                            SMBVDEBUG("set times error %d file %s\n",
1175                                error, np->n_rpath);
1176                    } else {
1177                            modified = 1;
1178                    }
1179            }
1180
1181    out:
1182            if (modified) {
1183                    /*
1184                     * Invalidate attribute cache in case the server
1185                     * doesn't set exactly the attributes we asked.
1186                     */
1187                    smbfs_attrcache_remove(np);
1188            }
1189
1190            if (have_fid) {
1191                    cerror = smbfs_smb_tmpclose(np, fid, &scred);
1192                    if (cerror)
1193                            SMBVDEBUG("error %d closing %s\n",
1194                                cerror, np->n_rpath);
1195            }
1196
1197            smb_credrele(&scred);
1198            smbfs_rw_exit(&np->r_lkserlock);
1199
1200            return (error);
1201    }
1202
1203    /*
1204     * smbfs_access_rwx()
1205     * Common function for smbfs_access, etc.
1206     *
1207     * The security model implemented by the FS is unusual
1208     * due to the current "single user mounts" restriction:
1209     * All access under a given mount point uses the CIFS
1210     * credentials established by the owner of the mount.
1211     *
1212     * Most access checking is handled by the CIFS server,
1213     * but we need sufficient Unix access checks here to
1214     * prevent other local Unix users from having access
1215     * to objects under this mount that the uid/gid/mode
1216     * settings in the mount would not allow.
1217     *
1218     * With this model, there is a case where we need the
1219     * ability to do an access check before we have the
1220     * vnode for an object.  This function takes advantage
1221     * of the fact that the uid/gid/mode is per mount, and
1222     * avoids the need for a vnode.
1223     *
1224     * We still (sort of) need a vnode when we call
1225     * secpolicy_vnode_access, but that only uses
1226     * the vtype field, so we can use a pair of fake
1227     * vnodes that have only v_type filled in.
1228     *
```

```
1229  * XXX: Later, add a new secpolicy_vtype_access()
1230  * that takes the vtype instead of a vnode, and
1231  * get rid of the tmpl_vxxx fake vnodes below.
1232  */
1233 static int
1234 smbfs_access_rwx(vfs_t *vfsp, int vtype, int mode, cred_t *cr)
1235 {
1236         /* See the secpolicy call below. */
1237         static const vnode_t tmpl_vdir = { .v_type = VDIR };
1238         static const vnode_t tmpl_vreg = { .v_type = VREG };
1239         vattr_t         va;
1240         vnode_t         *tvp;
1241         struct smbmntinfo *smi = VFTOSMI(vfsp);
1242         int shift = 0;

1244         /*
1245          * Build our (fabricated) vnode attributes.
1246          * XXX: Could make these templates in the
1247          * per-mount struct and use them here.
1248          */
1249         bzero(&va, sizeof (va));
1250         va.va_mask = AT_TYPE | AT_MODE | AT_UID | AT_GID;
1251         va.va_type = vtype;
1252         va.va_mode = (vtype == VDIR) ?
1253             smi->smi_dmode : smi->smi_fmode;
1254         va.va_uid = smi->smi_uid;
1255         va.va_gid = smi->smi_gid;

1257         /*
1258          * Disallow write attempts on read-only file systems,
1259          * unless the file is a device or fifo node.  Note:
1260          * Inline vn_is_readonly and IS_DEVVP here because
1261          * we may not have a vnode ptr.  Original expr. was:
1262          * (mode & VWRITE) && vn_is_readonly(vp) && !IS_DEVVP(vp))
1263          */
1264         if ((mode & VWRITE) &&
1265             (vfsp->vfs_flag & VFS_RDONLY) &&
1266             !(vtype == VCHR || vtype == VBLK || vtype == VFIFO))
1267                 return (EROFS);

1269         /*
1270          * Disallow attempts to access mandatory lock files.
1271          * Similarly, expand MANDLOCK here.
1272          * XXX: not sure we need this.
1273          */
1274         if ((mode & (VWRITE | VREAD | VEXEC)) &&
1275             va.va_type == VREG && MANDMODE(va.va_mode))
1276                 return (EACCES);

1278         /*
1279          * Access check is based on only
1280          * one of owner, group, public.
1281          * If not owner, then check group.
1282          * If not a member of the group,
1283          * then check public access.
1284          */
1285         if (crgetuid(cr) != va.va_uid) {
1286                 shift += 3;
1287                 if (!groupmember(va.va_gid, cr))
1288                         shift += 3;
1289         }

1291         /*
1292          * We need a vnode for secpolicy_vnode_access,
1293          * but the only thing it looks at is v_type,
1294          * so pass one of the templates above.
```



```
1295          */
1296         tvp = (va.va_type == VDIR) ?
1297             (vnode_t *)&tmpl_vdir :
1298             (vnode_t *)&tmpl_vreg;

1300         return (secpolicy_vnode_access2(cr, tvp, va.va_uid,
1301             va.va_mode << shift, mode));
1302 }

1304 /*
1305  * See smbfs_setattr
1306  */
1307 static int
1308 smbfs_accessx(void *arg, int mode, cred_t *cr)
1309 {
1310         vnode_t *vp = arg;
1311         /*
1312          * Note: The caller has checked the current zone,
1313          * the SMI_DEAD and VFS_UNMOUNTED flags, etc.
1314          */
1315         return (smbfs_access_rwx(vp->v_vfsp, vp->v_type, mode, cr));
1316 }

1318 /*
1319  * XXX
1320  * This op should support PSARC 2007/403, Modified Access Checks for CIFS
1321  */
1322 /* ARGSUSED */
1323 static int
1324 smbfs_access(vnode_t *vp, int mode, int flags, cred_t *cr, caller_context_t *ct)
1325 {
1326         vfs_t           *vfsp;
1327         smbmntinfo_t    *smi;

1329         vfsp = vp->v_vfsp;
1330         smi = VFTOSMI(vfsp);

1332         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1333                 return (EIO);

1335         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
1336                 return (EIO);

1338         return (smbfs_access_rwx(vfsp, vp->v_type, mode, cr));
1339 }


1342 /*
1343  * Flush local dirty pages to stable storage on the server.
1344  *
1345  * If FNODSYNC is specified, then there is nothing to do because
1346  * metadata changes are not cached on the client before being
1347  * sent to the server.
1348  */
1349 /* ARGSUSED */
1350 static int
1351 smbfs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
1352 {
1353         int             error = 0;
1354         smbmntinfo_t    *smi;
1355         smbnode_t       *np;
1356         struct smb_cred scred;

1358         np = VTOSMB(vp);
1359         smi = VTOSMI(vp);
```

```
1361         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1362                 return (EIO);
1363
1364         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1365                 return (EIO);
1366
1367         if ((syncflag & FNODSYNC) || IS_SWAPVP(vp))
1368                 return (0);
1369
1370         if ((syncflag & (FSYNC|FDSYNC)) == 0)
1371                 return (0);
1372
1373         /* Shared lock for n_fid use in _flush */
1374         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
1375                 return (EINTR);
1376         smb_credinit(&scred, cr);
1377
1378         error = smbfs_smb_flush(np, &scred);
1379
1380         smb_credrele(&scred);
1381         smbfs_rw_exit(&np->r_lkserlock);
1382
1383         return (error);
1384 }
1385
1386 /*
1387  * Last reference to vnode went away.
1388  */
1389 /* ARGSUSED */
1390 static void
1391 smbfs_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1392 {
1393         smbnode_t       *np;
1394         struct smb_cred scred;
1395
1396         /*
1397          * Don't "bail out" for VFS_UNMOUNTED here,
1398          * as we want to do cleanup, etc.
1399          * See also pcfs_inactive
1400          */
1401
1402         np = VTOSMB(vp);
1403
1404         /*
1405          * If this is coming from the wrong zone, we let someone in the right
1406          * zone take care of it asynchronously.  We can get here due to
1407          * VN_RELE() being called from pageout() or fsflush().  This call may
1408          * potentially turn into an expensive no-op if, for instance, v_count
1409          * gets incremented in the meantime, but it's still correct.
1410          */
1411
1412         /*
1413          * Defend against the possibility that higher-level callers
1414          * might not correctly balance open and close calls.  If we
1415          * get here with open references remaining, it means there
1416          * was a missing VOP_CLOSE somewhere.  If that happens, do
1417          * the close here so we don't "leak" FIDs on the server.
1418          *
1419          * Exclusive lock for modifying n_fid stuff.
1420          * Don't want this one ever interruptible.
1421          */
1422         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0);
1423         smb_credinit(&scred, cr);
1424
1425         switch (np->n_ovtype) {
1426         case VNON:
```



```
1427                 /* not open (OK) */
1428                 break;
1429
1430         case VDIR:
1431                 if (np->n_dirrefs == 0)
1432                         break;
1433                 SMBVDEBUG("open dir: refs %d path %s\n",
1434                     np->n_dirrefs, np->n_rpath);
1435                 /* Force last close. */
1436                 np->n_dirrefs = 1;
1437                 smbfs_rele_fid(np, &scred);
1438                 break;
1439
1440         case VREG:
1441                 if (np->n_fidrefs == 0)
1442                         break;
1443                 SMBVDEBUG("open file: refs %d id 0x%x path %s\n",
1444                     np->n_fidrefs, np->n_fid, np->n_rpath);
1445                 /*
1446                  * Before otW close, make sure dirty pages written back.
1447                  */
1448                 if (vn_has_cached_data(vp)) {
1449                         /* smbfs_putapage() will acquire shared lock, so release
1450                          * exclusive lock temporally.
1451                          */
1452                         smbfs_rw_exit(&np->r_lkserlock);
1453
1454                         (void) smbfs_putpage(vp, (offset_t) 0, 0, B_INVAL | B_AS
1455
1456                         /* acquire exclusive lock again. */
1457                         (void) smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, 0
1458                 }
1459 #endif /* ! codereview */
1460                 /* Force last close. */
1461                 np->n_fidrefs = 1;
1462                 smbfs_rele_fid(np, &scred);
1463                 break;
1464
1465         default:
1466                 SMBVDEBUG("bad n_ovtype %d\n", np->n_ovtype);
1467                 np->n_ovtype = VNON;
1468                 break;
1469         }
1470
1471         smb_credrele(&scred);
1472         smbfs_rw_exit(&np->r_lkserlock);
1473
1474         smbfs_addfree(np);
1475 }
1476
1477 /*
1478  * Remote file system operations having to do with directory manipulation.
1479  */
1480 /* ARGSUSED */
1481 static int
1482 smbfs_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
1483         int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
1484         int *direntflags, pathname_t *realpnp)
1485 {
1486         vfs_t           *vfs;
1487         smbmntinfo_t    *smi;
1488         smbnode_t       *dnp;
1489         int             error;
1490
1491         vfs = dvp->v_vfsp;
1492         smi = VFTOSMI(vfs);
```

```
1494         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1495                 return (EPERM);
1496
1497         if (smi->smi_flags & SMI_DEAD || vfs->vfs_flag & VFS_UNMOUNTED)
1498                 return (EIO);
1499
1500         dnp = VTOSMB(dvp);
1501
1502         /*
1503          * Are we looking up extended attributes?  If so, "dvp" is
1504          * the file or directory for which we want attributes, and
1505          * we need a lookup of the (faked up) attribute directory
1506          * before we lookup the rest of the path.
1507          */
1508         if (flags & LOOKUP_XATTR) {
1509                 /*
1510                  * Require the xattr mount option.
1511                  */
1512                 if ((vfs->vfs_flag & VFS_XATTR) == 0)
1513                         return (EINVAL);
1514
1515                 error = smbfs_get_xattrdir(dvp, vpp, cr, flags);
1516                 return (error);
1517         }
1518
1519         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_READER, SMBINTR(dvp)))
1520                 return (EINTR);
1521
1522         error = smbfslookup(dvp, nm, vpp, cr, 1, ct);
1523
1524         smbfs_rw_exit(&dnp->r_rwlock);
1525
1526         return (error);
1527 }
1528
1529 /* ARGSUSED */
1530 static int
1531 smbfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
1532         int cache_ok, caller_context_t *ct)
1533 {
1534         int             error;
1535         int             supplen; /* supported length */
1536         vnode_t         *vp;
1537         smbnode_t       *np;
1538         smbnode_t       *dnp;
1539         smbmntinfo_t    *smi;
1540         /* struct smb_vc        *vcp; */
1541         const char      *ill;
1542         const char      *name = (const char *)nm;
1543         int             nmlen = strlen(nm);
1544         int             rplen;
1545         struct smb_cred scred;
1546         struct smbfattr fa;
1547
1548         smi = VTOSMI(dvp);
1549         dnp = VTOSMB(dvp);
1550
1551         ASSERT(curproc->p_zone == smi->smi_zone_ref.zref_zone);
1552
1553 #ifdef NOT_YET
1554         vcp = SSTOVC(smi->smi_share);
1555
1556         /* XXX: Should compute this once and store it in smbmntinfo_t */
1557         supplen = (SMB_DIALECT(vcp) >= SMB_DIALECT_LANMAN2_0) ? 255 : 12;
1558 #else
```



```
1559         supplen = 255;
1560 #endif
1561
1562         /*
1563          * RWlock must be held, either reader or writer.
1564          * XXX: Can we check without looking directly
1565          * inside the struct smbfs_rwlock_t?
1566          */
1567         ASSERT(dnp->r_rwlock.count != 0);
1568
1569         /*
1570          * If lookup is for "", just return dvp.
1571          * No need to perform any access checks.
1572          */
1573         if (nmlen == 0) {
1574                 VN_HOLD(dvp);
1575                 *vpp = dvp;
1576                 return (0);
1577         }
1578
1579         /*
1580          * Can't do lookups in non-directories.
1581          */
1582         if (dvp->v_type != VDIR)
1583                 return (ENOTDIR);
1584
1585         /*
1586          * Need search permission in the directory.
1587          */
1588         error = smbfs_access(dvp, VEXEC, 0, cr, ct);
1589         if (error)
1590                 return (error);
1591
1592         /*
1593          * If lookup is for ".", just return dvp.
1594          * Access check was done above.
1595          */
1596         if (nmlen == 1 && name[0] == '.') {
1597                 VN_HOLD(dvp);
1598                 *vpp = dvp;
1599                 return (0);
1600         }
1601
1602         /*
1603          * Now some sanity checks on the name.
1604          * First check the length.
1605          */
1606         if (nmlen > supplen)
1607                 return (ENAMETOOLONG);
1608
1609         /*
1610          * Avoid surprises with characters that are
1611          * illegal in Windows file names.
1612          * Todo: CATIA mappings  XXX
1613          */
1614         ill = illegal_chars;
1615         if (dnp->n_flag & N_XATTR)
1616                 ill++; /* allow colon */
1617         if (strpbrk(nm, ill))
1618                 return (EINVAL);
1619
1620         /*
1621          * Special handling for lookup of ".."
1622          *
1623          * We keep full pathnames (as seen on the server)
1624          * so we can just trim off the last component to
```

```
1625          * get the full pathname of the parent.  Note:
1626          * We don't actually copy and modify, but just
1627          * compute the trimmed length and pass that with
1628          * the current dir path (not null terminated).
1629          *
1630          * We don't go over-the-wire to get attributes
1631          * for ".." because we know it's a directory,
1632          * and we can just leave the rest "stale"
1633          * until someone does a getattr.
1634          */
1635         if (nmlen == 2 && name[0] == '.' && name[1] == '.') {
1636                 if (dvp->v_flag & VROOT) {
1637                         /*
1638                          * Already at the root.  This can happen
1639                          * with directory listings at the root,
1640                          * which lookup "." and ".." to get the
1641                          * inode numbers.  Let ".." be the same
1642                          * as "." in the FS root.
1643                          */
1644                         VN_HOLD(dvp);
1645                         *vpp = dvp;
1646                         return (0);
1647                 }
1649                 /*
1650                  * Special case for XATTR directory
1651                  */
1652                 if (dvp->v_flag & V_XATTRDIR) {
1653                         error = smbfs_xa_parent(dvp, vpp);
1654                         return (error);
1655                 }
1657                 /*
1658                  * Find the parent path length.
1659                  */
1660                 rplen = dnp->n_rplen;
1661                 ASSERT(rplen > 0);
1662                 while (--rplen >= 0) {
1663                         if (dnp->n_rpath[rplen] == '\\')
1664                                 break;
1665                 }
1666                 if (rplen <= 0) {
1667                         /* Found our way to the root. */
1668                         vp = SMBTOV(smi->smi_root);
1669                         VN_HOLD(vp);
1670                         *vpp = vp;
1671                         return (0);
1672                 }
1673                 np = smbfs_node_findcreate(smi,
1674                     dnp->n_rpath, rplen, NULL, 0, 0,
1675                     &smbfs_fattr0); /* force create */
1676                 ASSERT(np != NULL);
1677                 vp = SMBTOV(np);
1678                 vp->v_type = VDIR;
1680                 /* Success! */
1681                 *vpp = vp;
1682                 return (0);
1683         }
1685         /*
1686          * Normal lookup of a name under this directory.
1687          * Note we handled "", ".", ".." above.
1688          */
1689         if (cache_ok) {
1690                 /*
```



```
1691                  * The caller indicated that it's OK to use a
1692                  * cached result for this lookup, so try to
1693                  * reclaim a node from the smbfs node cache.
1694                  */
1695                 error = smbfslookup_cache(dvp, nm, nmlen, &vp, cr);
1696                 if (error)
1697                         return (error);
1698                 if (vp != NULL) {
1699                         /* hold taken in lookup_cache */
1700                         *vpp = vp;
1701                         return (0);
1702                 }
1703         }
1705         /*
1706          * OK, go over-the-wire to get the attributes,
1707          * then create the node.
1708          */
1709         smb_credinit(&scred, cr);
1710         /* Note: this can allocate a new "name" */
1711         error = smbfs_smb_lookup(dnp, &name, &nmlen, &fa, &scred);
1712         smb_credrele(&scred);
1713         if (error == ENOTDIR) {
1714                 /*
1715                  * Lookup failed because this directory was
1716                  * removed or renamed by another client.
1717                  * Remove any cached attributes under it.
1718                  */
1719                 smbfs_attrcache_remove(dnp);
1720                 smbfs_attrcache_prune(dnp);
1721         }
1722         if (error)
1723                 goto out;
1725         error = smbfs_nget(dvp, name, nmlen, &fa, &vp);
1726         if (error)
1727                 goto out;
1729         /* Success! */
1730         *vpp = vp;
1732 out:
1733         /* smbfs_smb_lookup may have allocated name. */
1734         if (name != nm)
1735                 smbfs_name_free(name, nmlen);
1737         return (error);
1738 }
1740 /*
1741  * smbfslookup_cache
1742  *
1743  * Try to reclaim a node from the smbfs node cache.
1744  * Some statistics for DEBUG.
1745  *
1746  * This mechanism lets us avoid many of the five (or more)
1747  * OtW lookup calls per file seen with "ls -l" if we search
1748  * the smbfs node cache for recently inactive(ated) nodes.
1749  */
1750 #ifdef DEBUG
1751 int smbfs_lookup_cache_calls = 0;
1752 int smbfs_lookup_cache_error = 0;
1753 int smbfs_lookup_cache_miss = 0;
1754 int smbfs_lookup_cache_stale = 0;
1755 int smbfs_lookup_cache_hits = 0;
1756 #endif /* DEBUG */
```

```
1758 /* ARGSUSED */
1759 static int
1760 smbfslookup_cache(vnode_t *dvp, char *nm, int nmlen,
1761         vnode_t **vpp, cred_t *cr)
1762 {
1763         struct vattr va;
1764         smbnode_t *dnp;
1765         smbnode_t *np;
1766         vnode_t *vp;
1767         int error;
1768         char sep;

1770         dnp = VTOSMB(dvp);
1771         *vpp = NULL;

1773 #ifdef DEBUG
1774         smbfs_lookup_cache_calls++;
1775 #endif

1777         /*
1778          * First make sure we can get attributes for the
1779          * directory.  Cached attributes are OK here.
1780          * If we removed or renamed the directory, this
1781          * will return ENOENT.  If someone else removed
1782          * this directory or file, we'll find out when we
1783          * try to open or get attributes.
1784          */
1785         va.va_mask = AT_TYPE | AT_MODE;
1786         error = smbfsgetattr(dvp, &va, cr);
1787         if (error) {
1788 #ifdef DEBUG
1789                 smbfs_lookup_cache_error++;
1790 #endif
1791                 return (error);
1792         }

1794         /*
1795          * Passing NULL smbfattr here so we will
1796          * just look, not create.
1797          */
1798         sep = SMBFS_DNP_SEP(dnp);
1799         np = smbfs_node_findcreate(dnp->n_mount,
1800             dnp->n_rpath, dnp->n_rplen,
1801             nm, nmlen, sep, NULL);
1802         if (np == NULL) {
1803 #ifdef DEBUG
1804                 smbfs_lookup_cache_miss++;
1805 #endif
1806                 return (0);
1807         }

1809         /*
1810          * Found it.  Attributes still valid?
1811          */
1812         vp = SMBTOV(np);
1813         if (np->r_attrtime <= gethrtime()) {
1814                 /* stale */
1815 #ifdef DEBUG
1816                 smbfs_lookup_cache_stale++;
1817 #endif
1818                 VN_RELE(vp);
1819                 return (0);
1820         }

1822         /*
```



```
1823          * Success!
1824          * Caller gets hold from smbfs_node_findcreate
1825          */
1826 #ifdef DEBUG
1827         smbfs_lookup_cache_hits++;
1828 #endif
1829         *vpp = vp;
1830         return (0);
1831 }

1833 /*
1834  * XXX
1835  * vsecattr_t is new to build 77, and we need to eventually support
1836  * it in order to create an ACL when an object is created.
1837  *
1838  * This op should support the new FIGNORECASE flag for case-insensitive
1839  * lookups, per PSARC 2007/244.
1840  */
1841 /* ARGSUSED */
1842 static int
1843 smbfs_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
1844         int mode, vnode_t **vpp, cred_t *cr, int lfaware, caller_context_t *ct,
1845         vsecattr_t *vsecp)
1846 {
1847         int             error;
1848         int             cerror;
1849         vfs_t           *vfsp;
1850         vnode_t         *vp;
1851 #ifdef NOT_YET
1852         smbnode_t       *np;
1853 #endif
1854         smbnode_t       *dnp;
1855         smbmntinfo_t    *smi;
1856         struct vattr    vattr;
1857         struct smbfattr fattr;
1858         struct smb_cred scred;
1859         const char *name = (const char *)nm;
1860         int             nmlen = strlen(nm);
1861         uint32_t        disp;
1862         uint16_t        fid;
1863         int             xattr;

1865         vfsp = dvp->v_vfsp;
1866         smi = VFTOSMI(vfsp);
1867         dnp = VTOSMB(dvp);
1868         vp = NULL;

1870         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
1871                 return (EPERM);

1873         if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
1874                 return (EIO);

1876         /*
1877          * Note: this may break mknod(2) calls to create a directory,
1878          * but that's obscure use.  Some other filesystems do this.
1879          * XXX: Later, redirect VDIR type here to _mkdir.
1880          */
1881         if (va->va_type != VREG)
1882                 return (EINVAL);

1884         /*
1885          * If the pathname is "", just use dvp, no checks.
1886          * Do this outside of the rwlock (like zfs).
1887          */
1888         if (nmlen == 0) {
```

```
1889                VN_HOLD(dvp);
1890                *vpp = dvp;
1891                return (0);
1892        }

1894        /* Don't allow "." or ".." through here. */
1895        if ((nmlen == 1 && name[0] == '.') ||
1896            (nmlen == 2 && name[0] == '.' && name[1] == '.'))
1897                return (EISDIR);

1899        /*
1900         * We make a copy of the attributes because the caller does not
1901         * expect us to change what va points to.
1902         */
1903        vattr = *va;

1905        if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
1906                return (EINTR);
1907        smb_credinit(&scred, cr);

1909        /*
1910         * XXX: Do we need r_lkserlock too?
1911         * No use of any shared fid or fctx...
1912         */

1914        /*
1915         * NFS needs to go over the wire, just to be sure whether the
1916         * file exists or not.  Using a cached result is dangerous in
1917         * this case when making a decision regarding existence.
1918         *
1919         * The SMB protocol does NOT really need to go OTW here
1920         * thanks to the expressive NTCREATE disposition values.
1921         * Unfortunately, to do Unix access checks correctly,
1922         * we need to know if the object already exists.
1923         * When the object does not exist, we need VWRITE on
1924         * the directory.  Note: smbfslookup() checks VEXEC.
1925         */
1926        error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
1927        if (error == 0) {
1928                /*
1929                 * The file already exists.  Error?
1930                 * NB: have a hold from smbfslookup
1931                 */
1932                if (exclusive == EXCL) {
1933                        error = EEXIST;
1934                        VN_RELE(vp);
1935                        goto out;
1936                }
1937                /*
1938                 * Verify requested access.
1939                 */
1940                error = smbfs_access(vp, mode, 0, cr, ct);
1941                if (error) {
1942                        VN_RELE(vp);
1943                        goto out;
1944                }

1946                /*
1947                 * Truncate (if requested).
1948                 */
1949                if ((vattr.va_mask & AT_SIZE) && vattr.va_size == 0) {
1950                        vattr.va_mask = AT_SIZE;
1951                        error = smbfssetattr(vp, &vattr, 0, cr);
1952                        if (error) {
1953                                VN_RELE(vp);
1954                                goto out;
```



```
1955                        }
1956                }
1957                /* Success! */
1958 #ifdef NOT_YET
1959                vnevent_create(vp, ct);
1960 #endif
1961                *vpp = vp;
1962                goto out;
1963        }

1965        /*
1966         * The file did not exist.  Need VWRITE in the directory.
1967         */
1968        error = smbfs_access(dvp, VWRITE, 0, cr, ct);
1969        if (error)
1970                goto out;

1972        /*
1973         * Now things get tricky.  We also need to check the
1974         * requested open mode against the file we may create.
1975         * See comments at smbfs_access_rwx
1976         */
1977        error = smbfs_access_rwx(vfsp, VREG, mode, cr);
1978        if (error)
1979                goto out;

1981        /*
1982         * Now the code derived from Darwin,
1983         * but with greater use of NT_CREATE
1984         * disposition options.  Much changed.
1985         *
1986         * Create (or open) a new child node.
1987         * Note we handled "." and ".." above.
1988         */

1990        if (exclusive == EXCL)
1991                disp = NTCREATEX_DISP_CREATE;
1992        else {
1993                /* Truncate regular files if requested. */
1994                if ((va->va_type == VREG) &&
1995                    (va->va_mask & AT_SIZE) &&
1996                    (va->va_size == 0))
1997                        disp = NTCREATEX_DISP_OVERWRITE_IF;
1998                else
1999                        disp = NTCREATEX_DISP_OPEN_IF;
2000        }
2001        xattr = (dnp->n_flag & N_XATTR) ? 1 : 0;
2002        error = smbfs_smb_create(dnp,
2003            name, nmlen, xattr,
2004            disp, &scred, &fid);
2005        if (error)
2006                goto out;

2008        /*
2009         * XXX: Missing some code here to deal with
2010         * the case where we opened an existing file,
2011         * it's size is larger than 32-bits, and we're
2012         * setting the size from a process that's not
2013         * aware of large file offsets.  i.e.
2014         * from the NFS3 code:
2015         */
2016 #if NOT_YET /* XXX */
2017        if ((vattr.va_mask & AT_SIZE) &&
2018            vp->v_type == VREG) {
2019                np = VTOSMB(vp);
2020                /*
```

```
2021                 * Check here for large file handled
2022                 * by LF-unaware process (as
2023                 * ufs_create() does)
2024                 */
2025                if (!(lfaware & FOFFMAX)) {
2026                        mutex_enter(&np->r_statelock);
2027                        if (np->r_size > MAXOFF32_T)
2028                                error = EOVERFLOW;
2029                        mutex_exit(&np->r_statelock);
2030                }
2031                if (!error) {
2032                        vattr.va_mask = AT_SIZE;
2033                        error = smbfssetattr(vp,
2034                            &vattr, 0, cr);
2035                }
2036        }
2037 #endif /* XXX */
2038        /*
2039         * Should use the fid to get/set the size
2040         * while we have it opened here.  See above.
2041         */

2043        cerror = smbfs_smb_close(smi->smi_share, fid, NULL, &scred);
2044        if (cerror)
2045                SMBVDEBUG("error %d closing %s\\%s\n",
2046                    cerror, dnp->n_rpath, name);

2048        /*
2049         * In the open case, the name may differ a little
2050         * from what we passed to create (case, etc.)
2051         * so call lookup to get the (opened) name.
2052         *
2053         * XXX: Could avoid this extra lookup if the
2054         * "createact" result from NT_CREATE says we
2055         * created the object.
2056         */
2057        error = smbfs_smb_lookup(dnp, &name, &nmlen, &fattr, &scred);
2058        if (error)
2059                goto out;

2061        /* update attr and directory cache */
2062        smbfs_attr_touchdir(dnp);

2064        error = smbfs_nget(dvp, name, nmlen, &fattr, &vp);
2065        if (error)
2066                goto out;

2068        /* XXX invalidate pages if we truncated? */

2070        /* Success! */
2071        *vpp = vp;
2072        error = 0;

2074 out:
2075        smb_credrele(&scred);
2076        smbfs_rw_exit(&dnp->r_rwlock);
2077        if (name != nm)
2078                smbfs_name_free(name, nmlen);
2079        return (error);
2080 }

2082 /*
2083  * XXX
2084  * This op should support the new FIGNORECASE flag for case-insensitive
2085  * lookups, per PSARC 2007/244.
2086  */
```



```
2087 /* ARGSUSED */
2088 static int
2089 smbfs_remove(vnode_t *dvp, char *nm, cred_t *cr, caller_context_t *ct,
2090        int flags)
2091 {
2092        int             error;
2093        vnode_t         *vp;
2094        smbnode_t       *np;
2095        smbnode_t       *dnp;
2096        struct smb_cred scred;
2097        /* enum smbfsstat status; */
2098        smbmntinfo_t    *smi;

2100        smi = VTOSMI(dvp);

2102        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2103                return (EPERM);

2105        if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2106                return (EIO);

2108        dnp = VTOSMB(dvp);
2109        if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2110                return (EINTR);
2111        smb_credinit(&scred, cr);

2113        /*
2114         * Verify access to the dirctory.
2115         */
2116        error = smbfs_access(dvp, VWRITE|VEXEC, 0, cr, ct);
2117        if (error)
2118                goto out;

2120        /*
2121         * NOTE:  the darwin code gets the "vp" passed in so it looks
2122         * like the "vp" has probably been "lookup"ed by the VFS layer.
2123         * It looks like we will need to lookup the vp to check the
2124         * caches and check if the object being deleted is a directory.
2125         */
2126        error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2127        if (error)
2128                goto out;

2130        /* Never allow link/unlink directories on CIFS. */
2131        if (vp->v_type == VDIR) {
2132                VN_RELE(vp);
2133                error = EPERM;
2134                goto out;
2135        }

2137        /*
2138         * Now we have the real reference count on the vnode
2139         * Do we have the file open?
2140         */
2141        np = VTOSMB(vp);
2142        mutex_enter(&np->r_statelock);
2143        if ((vp->v_count > 1) && (np->n_fidrefs > 0)) {
2144                /*
2145                 * NFS does a rename on remove here.
2146                 * Probably not applicable for SMB.
2147                 * Like Darwin, just return EBUSY.
2148                 *
2149                 * XXX: Todo - Use Trans2rename, and
2150                 * if that fails, ask the server to
2151                 * set the delete-on-close flag.
2152                 */
```

```
2153                    mutex_exit(&np->r_statelock);
2154                    error = EBUSY;
2155            } else {
2156                    smbfs_attrcache_rm_locked(np);
2157                    mutex_exit(&np->r_statelock);
2159                    error = smbfs_smb_delete(np, &scred, NULL, 0, 0);
2161                    /*
2162                     * If the file should no longer exist, discard
2163                     * any cached attributes under this node.
2164                     */
2165                    switch (error) {
2166                    case 0:
2167                    case ENOENT:
2168                    case ENOTDIR:
2169                            smbfs_attrcache_prune(np);
2170                            break;
2171                    }
2172            }
2174            VN_RELE(vp);
2176    out:
2177            smb_credrele(&scred);
2178            smbfs_rw_exit(&dnp->r_rwlock);
2180            return (error);
2181    }
2184    /*
2185     * XXX
2186     * This op should support the new FIGNORECASE flag for case-insensitive
2187     * lookups, per PSARC 2007/244.
2188     */
2189    /* ARGSUSED */
2190    static int
2191    smbfs_rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
2192            caller_context_t *ct, int flags)
2193    {
2194            /* vnode_t              *realvp; */
2196            if (curproc->p_zone != VTOSMI(odvp)->smi_zone_ref.zref_zone ||
2197                curproc->p_zone != VTOSMI(ndvp)->smi_zone_ref.zref_zone)
2198                    return (EPERM);
2200            if (VTOSMI(odvp)->smi_flags & SMI_DEAD ||
2201                VTOSMI(ndvp)->smi_flags & SMI_DEAD ||
2202                odvp->v_vfsp->vfs_flag & VFS_UNMOUNTED ||
2203                ndvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2204                    return (EIO);
2206            return (smbfsrename(odvp, onm, ndvp, nnm, cr, ct));
2207    }
2209    /*
2210     * smbfsrename does the real work of renaming in SMBFS
2211     */
2212    /* ARGSUSED */
2213    static int
2214    smbfsrename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
2215            caller_context_t *ct)
2216    {
2217            int             error;
2218            int             nvp_locked = 0;
```



```
2219            vnode_t         *nvp = NULL;
2220            vnode_t         *ovp = NULL;
2221            smbnode_t       *onp;
2222            smbnode_t       *nnp;
2223            smbnode_t       *odnp;
2224            smbnode_t       *ndnp;
2225            struct smb_cred scred;
2226            /* enum smbfsstat       status; */
2228            ASSERT(curproc->p_zone == VTOSMI(odvp)->smi_zone_ref.zref_zone);
2230            if (strcmp(onm, ".") == 0 || strcmp(onm, "..") == 0 ||
2231                strcmp(nnm, ".") == 0 || strcmp(nnm, "..") == 0)
2232                    return (EINVAL);
2234            /*
2235             * Check that everything is on the same filesystem.
2236             * vn_rename checks the fsid's, but in case we don't
2237             * fill those in correctly, check here too.
2238             */
2239            if (odvp->v_vfsp != ndvp->v_vfsp)
2240                    return (EXDEV);
2242            odnp = VTOSMB(odvp);
2243            ndnp = VTOSMB(ndvp);
2245            /*
2246             * Avoid deadlock here on old vs new directory nodes
2247             * by always taking the locks in order of address.
2248             * The order is arbitrary, but must be consistent.
2249             */
2250            if (odnp < ndnp) {
2251                    if (smbfs_rw_enter_sig(&odnp->r_rwlock, RW_WRITER,
2252                        SMBINTR(odvp)))
2253                            return (EINTR);
2254                    if (smbfs_rw_enter_sig(&ndnp->r_rwlock, RW_WRITER,
2255                        SMBINTR(ndvp))) {
2256                            smbfs_rw_exit(&odnp->r_rwlock);
2257                            return (EINTR);
2258                    }
2259            } else {
2260                    if (smbfs_rw_enter_sig(&ndnp->r_rwlock, RW_WRITER,
2261                        SMBINTR(ndvp)))
2262                            return (EINTR);
2263                    if (smbfs_rw_enter_sig(&odnp->r_rwlock, RW_WRITER,
2264                        SMBINTR(odvp))) {
2265                            smbfs_rw_exit(&ndnp->r_rwlock);
2266                            return (EINTR);
2267                    }
2268            }
2269            smb_credinit(&scred, cr);
2270            /*
2271             * No returns after this point (goto out)
2272             */
2274            /*
2275             * Need write access on source and target.
2276             * Server takes care of most checks.
2277             */
2278            error = smbfs_access(odvp, VWRITE|VEXEC, 0, cr, ct);
2279            if (error)
2280                    goto out;
2281            if (odvp != ndvp) {
2282                    error = smbfs_access(ndvp, VWRITE, 0, cr, ct);
2283                    if (error)
2284                            goto out;
```

```
2285        }
2286
2287        /*
2288         * Lookup the source name.  Must already exist.
2289         */
2290        error = smbfslookup(odvp, onm, &ovp, cr, 0, ct);
2291        if (error)
2292                goto out;
2293
2294        /*
2295         * Lookup the target file.  If it exists, it needs to be
2296         * checked to see whether it is a mount point and whether
2297         * it is active (open).
2298         */
2299        error = smbfslookup(ndvp, nnm, &nvp, cr, 0, ct);
2300        if (!error) {
2301                /*
2302                 * Target (nvp) already exists.  Check that it
2303                 * has the same type as the source.  The server
2304                 * will check this also, (and more reliably) but
2305                 * this lets us return the correct error codes.
2306                 */
2307                if (ovp->v_type == VDIR) {
2308                        if (nvp->v_type != VDIR) {
2309                                error = ENOTDIR;
2310                                goto out;
2311                        }
2312                } else {
2313                        if (nvp->v_type == VDIR) {
2314                                error = EISDIR;
2315                                goto out;
2316                        }
2317                }
2318
2319                /*
2320                 * POSIX dictates that when the source and target
2321                 * entries refer to the same file object, rename
2322                 * must do nothing and exit without error.
2323                 */
2324                if (ovp == nvp) {
2325                        error = 0;
2326                        goto out;
2327                }
2328
2329                /*
2330                 * Also must ensure the target is not a mount point,
2331                 * and keep mount/umount away until we're done.
2332                 */
2333                if (vn_vfsrlock(nvp)) {
2334                        error = EBUSY;
2335                        goto out;
2336                }
2337                nvp_locked = 1;
2338                if (vn_mountedvfs(nvp) != NULL) {
2339                        error = EBUSY;
2340                        goto out;
2341                }
2342
2343                /*
2344                 * CIFS gives a SHARING_VIOLATION error when
2345                 * trying to rename onto an exising object,
2346                 * so try to remove the target first.
2347                 * (Only for files, not directories.)
2348                 */
2349                if (nvp->v_type == VDIR) {
2350                        error = EEXIST;
```



```
2351                        goto out;
2352                }
2353
2354                /*
2355                 * Nodes that are "not active" here have v_count=2
2356                 * because vn_renameat (our caller) did a lookup on
2357                 * both the source and target before this call.
2358                 * Otherwise this similar to smbfs_remove.
2359                 */
2360                nnp = VTOSMB(nvp);
2361                mutex_enter(&nnp->r_statelock);
2362                if ((nvp->v_count > 2) && (nnp->n_fidrefs > 0)) {
2363                        /*
2364                         * The target file exists, is not the same as
2365                         * the source file, and is active.  Other FS
2366                         * implementations unlink the target here.
2367                         * For SMB, we don't assume we can remove an
2368                         * open file.  Return an error instead.
2369                         */
2370                        mutex_exit(&nnp->r_statelock);
2371                        error = EBUSY;
2372                        goto out;
2373                }
2374
2375                /*
2376                 * Target file is not active. Try to remove it.
2377                 */
2378                smbfs_attrcache_rm_locked(nnp);
2379                mutex_exit(&nnp->r_statelock);
2380
2381                error = smbfs_smb_delete(nnp, &scred, NULL, 0, 0);
2382
2383                /*
2384                 * Similar to smbfs_remove
2385                 */
2386                switch (error) {
2387                case 0:
2388                case ENOENT:
2389                case ENOTDIR:
2390                        smbfs_attrcache_prune(nnp);
2391                        break;
2392                }
2393
2394                if (error)
2395                        goto out;
2396                /*
2397                 * OK, removed the target file.  Continue as if
2398                 * lookup target had failed (nvp == NULL).
2399                 */
2400                vn_vfsunlock(nvp);
2401                nvp_locked = 0;
2402                VN_RELE(nvp);
2403                nvp = NULL;
2404        } /* nvp */
2405
2406        onp = VTOSMB(ovp);
2407        smbfs_attrcache_remove(onp);
2408
2409        error = smbfs_smb_rename(onp, ndnp, nnm, strlen(nnm), &scred);
2410
2411        /*
2412         * If the old name should no longer exist,
2413         * discard any cached attributes under it.
2414         */
2415        if (error == 0)
2416                smbfs_attrcache_prune(onp);
```

```
2418 out:
2419         if (nvp) {
2420                 if (nvp_locked)
2421                         vn_vfsunlock(nvp);
2422                 VN_RELE(nvp);
2423         }
2424         if (ovp)
2425                 VN_RELE(ovp);

2427         smb_credrele(&scred);
2428         smbfs_rw_exit(&odnp->r_rwlock);
2429         smbfs_rw_exit(&ndnp->r_rwlock);

2431         return (error);
2432 }

2434 /*
2435  * XXX
2436  * vsecattr_t is new to build 77, and we need to eventually support
2437  * it in order to create an ACL when an object is created.
2438  *
2439  * This op should support the new FIGNORECASE flag for case-insensitive
2440  * lookups, per PSARC 2007/244.
2441  */
2442 /* ARGSUSED */
2443 static int
2444 smbfs_mkdir(vnode_t *dvp, char *nm, struct vattr *va, vnode_t **vpp,
2445         cred_t *cr, caller_context_t *ct, int flags, vsecattr_t *vsecp)
2446 {
2447         vnode_t         *vp;
2448         struct smbnode  *dnp = VTOSMB(dvp);
2449         struct smbmntinfo *smi = VTOSMI(dvp);
2450         struct smb_cred scred;
2451         struct smbfattr fattr;
2452         const char              *name = (const char *) nm;
2453         int             nmlen = strlen(name);
2454         int             error, hiderr;

2456         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2457                 return (EPERM);

2459         if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2460                 return (EIO);

2462         if ((nmlen == 1 && name[0] == '.') ||
2463             (nmlen == 2 && name[0] == '.' && name[1] == '.'))
2464                 return (EEXIST);

2466         /* Only plain files are allowed in V_XATTRDIR. */
2467         if (dvp->v_flag & V_XATTRDIR)
2468                 return (EINVAL);

2470         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2471                 return (EINTR);
2472         smb_credinit(&scred, cr);

2474         /*
2475          * XXX: Do we need r_lkserlock too?
2476          * No use of any shared fid or fctx...
2477          */

2479         /*
2480          * Require write access in the containing directory.
2481          */
2482         error = smbfs_access(dvp, VWRITE, 0, cr, ct);
```



```
2483         if (error)
2484                 goto out;

2486         error = smbfs_smb_mkdir(dnp, name, nmlen, &scred);
2487         if (error)
2488                 goto out;

2490         error = smbfs_smb_lookup(dnp, &name, &nmlen, &fattr, &scred);
2491         if (error)
2492                 goto out;

2494         smbfs_attr_touchdir(dnp);

2496         error = smbfs_nget(dvp, name, nmlen, &fattr, &vp);
2497         if (error)
2498                 goto out;

2500         if (name[0] == '.')
2501                 if ((hiderr = smbfs_smb_hideit(VTOSMB(vp), NULL, 0, &scred)))
2502                         SMBVDEBUG("hide failure %d\n", hiderr);

2504         /* Success! */
2505         *vpp = vp;
2506         error = 0;
2507 out:
2508         smb_credrele(&scred);
2509         smbfs_rw_exit(&dnp->r_rwlock);

2511         if (name != nm)
2512                 smbfs_name_free(name, nmlen);

2514         return (error);
2515 }

2517 /*
2518  * XXX
2519  * This op should support the new FIGNORECASE flag for case-insensitive
2520  * lookups, per PSARC 2007/244.
2521  */
2522 /* ARGSUSED */
2523 static int
2524 smbfs_rmdir(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
2525         caller_context_t *ct, int flags)
2526 {
2527         vnode_t         *vp = NULL;
2528         int             vp_locked = 0;
2529         struct smbmntinfo *smi = VTOSMI(dvp);
2530         struct smbnode  *dnp = VTOSMB(dvp);
2531         struct smbnode  *np;
2532         struct smb_cred scred;
2533         int             error;

2535         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2536                 return (EPERM);

2538         if (smi->smi_flags & SMI_DEAD || dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2539                 return (EIO);

2541         if (smbfs_rw_enter_sig(&dnp->r_rwlock, RW_WRITER, SMBINTR(dvp)))
2542                 return (EINTR);
2543         smb_credinit(&scred, cr);

2545         /*
2546          * Require w/x access in the containing directory.
2547          * Server handles all other access checks.
2548          */
```

```
2549         error = smbfs_access(dvp, VEXEC|VWRITE, 0, cr, ct);
2550         if (error)
2551                 goto out;

2553         /*
2554          * First lookup the entry to be removed.
2555          */
2556         error = smbfslookup(dvp, nm, &vp, cr, 0, ct);
2557         if (error)
2558                 goto out;
2559         np = VTOSMB(vp);

2561         /*
2562          * Disallow rmdir of "." or current dir, or the FS root.
2563          * Also make sure it's a directory, not a mount point,
2564          * and lock to keep mount/umount away until we're done.
2565          */
2566         if ((vp == dvp) || (vp == cdir) || (vp->v_flag & VROOT)) {
2567                 error = EINVAL;
2568                 goto out;
2569         }
2570         if (vp->v_type != VDIR) {
2571                 error = ENOTDIR;
2572                 goto out;
2573         }
2574         if (vn_vfsrlock(vp)) {
2575                 error = EBUSY;
2576                 goto out;
2577         }
2578         vp_locked = 1;
2579         if (vn_mountedvfs(vp) != NULL) {
2580                 error = EBUSY;
2581                 goto out;
2582         }

2584         smbfs_attrcache_remove(np);
2585         error = smbfs_smb_rmdir(np, &scred);

2587         /*
2588          * Similar to smbfs_remove
2589          */
2590         switch (error) {
2591         case 0:
2592         case ENOENT:
2593         case ENOTDIR:
2594                 smbfs_attrcache_prune(np);
2595                 break;
2596         }

2598         if (error)
2599                 goto out;

2601         mutex_enter(&np->r_statelock);
2602         dnp->n_flag |= NMODIFIED;
2603         mutex_exit(&np->r_statelock);
2604         smbfs_attr_touchdir(dnp);
2605         smbfs_rmhash(np);

2607 out:
2608         if (vp) {
2609                 if (vp_locked)
2610                         vn_vfsunlock(vp);
2611                 VN_RELE(vp);
2612         }
2613         smb_credrele(&scred);
2614         smbfs_rw_exit(&dnp->r_rwlock);
```



```
2616         return (error);
2617 }


2620 /* ARGSUSED */
2621 static int
2622 smbfs_readdir(vnode_t *vp, struct uio *uiop, cred_t *cr, int *eofp,
2623         caller_context_t *ct, int flags)
2624 {
2625         struct smbnode  *np = VTOSMB(vp);
2626         int             error = 0;
2627         smbmntinfo_t    *smi;

2629         smi = VTOSMI(vp);

2631         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2632                 return (EIO);

2634         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2635                 return (EIO);

2637         /*
2638          * Require read access in the directory.
2639          */
2640         error = smbfs_access(vp, VREAD, 0, cr, ct);
2641         if (error)
2642                 return (error);

2644         ASSERT(smbfs_rw_lock_held(&np->r_rwlock, RW_READER));

2646         /*
2647          * XXX: Todo readdir cache here
2648          * Note: NFS code is just below this.
2649          *
2650          * I am serializing the entire readdir opreation
2651          * now since we have not yet implemented readdir
2652          * cache. This fix needs to be revisited once
2653          * we implement readdir cache.
2654          */
2655         if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
2656                 return (EINTR);

2658         error = smbfs_readvdir(vp, uiop, cr, eofp, ct);

2660         smbfs_rw_exit(&np->r_lkserlock);

2662         return (error);
2663 }

2665 /* ARGSUSED */
2666 static int
2667 smbfs_readvdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
2668         caller_context_t *ct)
2669 {
2670         /*
2671          * Note: "limit" tells the SMB-level FindFirst/FindNext
2672          * functions how many directory entries to request in
2673          * each OtW call.  It needs to be large enough so that
2674          * we don't make lots of tiny OtW requests, but there's
2675          * no point making it larger than the maximum number of
2676          * OtW entries that would fit in a maximum sized trans2
2677          * response (64k / 48).  Beyond that, it's just tuning.
2678          * WinNT used 512, Win2k used 1366.  We use 1000.
2679          */
2680         static const int limit = 1000;
```

```
2681         /* Largest possible dirent size. */
2682         static const size_t dbufsiz = DIRENT64_RECLEN(SMB_MAXFNAMELEN);
2683         struct smb_cred scred;
2684         vnode_t         *newvp;
2685         struct smbnode  *np = VTOSMB(vp);
2686         struct smbfs_fctx *ctx;
2687         struct dirent64 *dp;
2688         ssize_t         save_resid;
2689         offset_t        save_offset; /* 64 bits */
2690         int             offset; /* yes, 32 bits */
2691         int             nmlen, error;
2692         ushort_t        reclen;

2694         ASSERT(curproc->p_zone == VTOSMI(vp)->smi_zone_ref.zref_zone);

2696         /* Make sure we serialize for n_dirseq use. */
2697         ASSERT(smbfs_rw_lock_held(&np->r_lkserlock, RW_WRITER));

2699         /*
2700          * Make sure smbfs_open filled in n_dirseq
2701          */
2702         if (np->n_dirseq == NULL)
2703                 return (EBADF);

2705         /* Check for overflow of (32-bit) directory offset. */
2706         if (uio->uio_loffset < 0 || uio->uio_loffset > INT32_MAX ||
2707             (uio->uio_loffset + uio->uio_resid) > INT32_MAX)
2708                 return (EINVAL);

2710         /* Require space for at least one dirent. */
2711         if (uio->uio_resid < dbufsiz)
2712                 return (EINVAL);

2714         SMBVDEBUG("dirname='%s'\n", np->n_rpath);
2715         smb_credinit(&scred, cr);
2716         dp = kmem_alloc(dbufsiz, KM_SLEEP);

2718         save_resid = uio->uio_resid;
2719         save_offset = uio->uio_loffset;
2720         offset = uio->uio_offset;
2721         SMBVDEBUG("in: offset=%d, resid=%d\n",
2722             (int)uio->uio_offset, (int)uio->uio_resid);
2723         error = 0;

2725         /*
2726          * Generate the "." and ".." entries here so we can
2727          * (1) make sure they appear (but only once), and
2728          * (2) deal with getting their I numbers which the
2729          * findnext below does only for normal names.
2730          */
2731         while (offset < FIRST_DIROFS) {
2732                 /*
2733                  * Tricky bit filling in the first two:
2734                  * offset 0 is ".", offset 1 is ".."
2735                  * so strlen of these is offset+1.
2736                  */
2737                 reclen = DIRENT64_RECLEN(offset + 1);
2738                 if (uio->uio_resid < reclen)
2739                         goto out;
2740                 bzero(dp, reclen);
2741                 dp->d_reclen = reclen;
2742                 dp->d_name[0] = '.';
2743                 dp->d_name[1] = '.';
2744                 dp->d_name[offset + 1] = '\0';
2745                 /*
2746                  * Want the real I-numbers for the "." and ".."
```



```
2747                  * entries.  For these two names, we know that
2748                  * smbfslookup can get the nodes efficiently.
2749                  */
2750                 error = smbfslookup(vp, dp->d_name, &newvp, cr, 1, ct);
2751                 if (error) {
2752                         dp->d_ino = np->n_ino + offset; /* fiction */
2753                 } else {
2754                         dp->d_ino = VTOSMB(newvp)->n_ino;
2755                         VN_RELE(newvp);
2756                 }
2757                 /*
2758                  * Note: d_off is the offset that a user-level program
2759                  * should seek to for reading the NEXT directory entry.
2760                  * See libc: readdir, telldir, seekdir
2761                  */
2762                 dp->d_off = offset + 1;
2763                 error = uiomove(dp, reclen, UIO_READ, uio);
2764                 if (error)
2765                         goto out;
2766                 /*
2767                  * Note: uiomove updates uio->uio_offset,
2768                  * but we want it to be our "cookie" value,
2769                  * which just counts dirents ignoring size.
2770                  */
2771                 uio->uio_offset = ++offset;
2772         }

2774         /*
2775          * If there was a backward seek, we have to reopen.
2776          */
2777         if (offset < np->n_dirofs) {
2778                 SMBVDEBUG("Reopening search %d:%d\n",
2779                     offset, np->n_dirofs);
2780                 error = smbfs_smb_findopen(np, "*", 1,
2781                     SMB_FA_SYSTEM | SMB_FA_HIDDEN | SMB_FA_DIR,
2782                     &scred, &ctx);
2783                 if (error) {
2784                         SMBVDEBUG("can not open search, error = %d", error);
2785                         goto out;
2786                 }
2787                 /* free the old one */
2788                 (void) smbfs_smb_findclose(np->n_dirseq, &scred);
2789                 /* save the new one */
2790                 np->n_dirseq = ctx;
2791                 np->n_dirofs = FIRST_DIROFS;
2792         } else {
2793                 ctx = np->n_dirseq;
2794         }

2796         /*
2797          * Skip entries before the requested offset.
2798          */
2799         while (np->n_dirofs < offset) {
2800                 error = smbfs_smb_findnext(ctx, limit, &scred);
2801                 if (error != 0)
2802                         goto out;
2803                 np->n_dirofs++;
2804         }

2806         /*
2807          * While there's room in the caller's buffer:
2808          *      get a directory entry from SMB,
2809          *      convert to a dirent, copyout.
2810          * We stop when there is no longer room for a
2811          * maximum sized dirent because we must decide
2812          * before we know anything about the next entry.
```

```
2813          */
2814         while (uio->uio_resid >= dbufsiz) {
2815                 error = smbfs_smb_findnext(ctx, limit, &scred);
2816                 if (error != 0)
2817                         goto out;
2818                 np->n_dirofs++;

2820                 /* Sanity check the name length. */
2821                 nmlen = ctx->f_nmlen;
2822                 if (nmlen > SMB_MAXFNAMELEN) {
2823                         nmlen = SMB_MAXFNAMELEN;
2824                         SMBVDEBUG("Truncating name: %s\n", ctx->f_name);
2825                 }
2826                 if (smbfs_fastlookup) {
2827                         /* See comment at smbfs_fastlookup above. */
2828                         if (smbfs_nget(vp, ctx->f_name, nmlen,
2829                             &ctx->f_attr, &newvp) == 0)
2830                                 VN_RELE(newvp);
2831                 }

2833                 reclen = DIRENT64_RECLEN(nmlen);
2834                 bzero(dp, reclen);
2835                 dp->d_reclen = reclen;
2836                 bcopy(ctx->f_name, dp->d_name, nmlen);
2837                 dp->d_name[nmlen] = '\0';
2838                 dp->d_ino = ctx->f_inum;
2839                 dp->d_off = offset + 1; /* See d_off comment above */
2840                 error = uiomove(dp, reclen, UIO_READ, uio);
2841                 if (error)
2842                         goto out;
2843                 /* See comment re. uio_offset above. */
2844                 uio->uio_offset = ++offset;
2845         }

2847 out:
2848         /*
2849          * When we come to the end of a directory, the
2850          * SMB-level functions return ENOENT, but the
2851          * caller is not expecting an error return.
2852          *
2853          * Also note that we must delay the call to
2854          * smbfs_smb_findclose(np->n_dirseq, ...)
2855          * until smbfs_close so that all reads at the
2856          * end of the directory will return no data.
2857          */
2858         if (error == ENOENT) {
2859                 error = 0;
2860                 if (eofp)
2861                         *eofp = 1;
2862         }
2863         /*
2864          * If we encountered an error (i.e. "access denied")
2865          * from the FindFirst call, we will have copied out
2866          * the "." and ".." entries leaving offset == 2.
2867          * In that case, restore the original offset/resid
2868          * so the caller gets no data with the error.
2869          */
2870         if (error != 0 && offset == FIRST_DIROFS) {
2871                 uio->uio_loffset = save_offset;
2872                 uio->uio_resid = save_resid;
2873         }
2874         SMBVDEBUG("out: offset=%d, resid=%d\n",
2875             (int)uio->uio_offset, (int)uio->uio_resid);

2877         kmem_free(dp, dbufsiz);
2878         smb_credrele(&scred);
```



```
2879         return (error);
2880 }


2883 /*
2884  * The pair of functions VOP_RWLOCK, VOP_RWUNLOCK
2885  * are optional functions that are called by:
2886  *    getdents, before/after VOP_READDIR
2887  *    pread, before/after ... VOP_READ
2888  *    pwrite, before/after ... VOP_WRITE
2889  *    (other places)
2890  *
2891  * Careful here: None of the above check for any
2892  * error returns from VOP_RWLOCK / VOP_RWUNLOCK!
2893  * In fact, the return value from _rwlock is NOT
2894  * an error code, but V_WRITELOCK_TRUE / _FALSE.
2895  *
2896  * Therefore, it's up to _this_ code to make sure
2897  * the lock state remains balanced, which means
2898  * we can't "bail out" on interrupts, etc.
2899  */

2901 /* ARGSUSED2 */
2902 static int
2903 smbfs_rwlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
2904 {
2905         smbnode_t       *np = VTOSMB(vp);

2907         if (!write_lock) {
2908                 (void) smbfs_rw_enter_sig(&np->r_rwlock, RW_READER, FALSE);
2909                 return (V_WRITELOCK_FALSE);
2910         }


2913         (void) smbfs_rw_enter_sig(&np->r_rwlock, RW_WRITER, FALSE);
2914         return (V_WRITELOCK_TRUE);
2915 }

2917 /* ARGSUSED */
2918 static void
2919 smbfs_rwunlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
2920 {
2921         smbnode_t       *np = VTOSMB(vp);

2923         smbfs_rw_exit(&np->r_rwlock);
2924 }


2927 /* ARGSUSED */
2928 static int
2929 smbfs_seek(vnode_t *vp, offset_t ooff, offset_t *noffp, caller_context_t *ct)
2930 {
2931         smbmntinfo_t    *smi;

2933         smi = VTOSMI(vp);

2935         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2936                 return (EPERM);

2938         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2939                 return (EIO);

2941         /*
2942          * Because we stuff the readdir cookie into the offset field
2943          * someone may attempt to do an lseek with the cookie which
2944          * we want to succeed.
```

```
2945          */
2946         if (vp->v_type == VDIR)
2947                 return (0);
2948
2949         /* Like NFS3, just check for 63-bit overflow. */
2950         if (*noffp < 0)
2951                 return (EINVAL);
2952
2953         return (0);
2954 }
2955
2956
2957 /*
2958  * XXX
2959  * This op may need to support PSARC 2007/440, nbmand changes for CIFS Service.
2960  */
2961 static int
2962 smbfs_frlock(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
2963         offset_t offset, struct flk_callback *flk_cbp, cred_t *cr,
2964         caller_context_t *ct)
2965 {
2966         if (curproc->p_zone != VTOSMI(vp)->smi_zone_ref.zref_zone)
2967                 return (EIO);
2968
2969         if (VTOSMI(vp)->smi_flags & SMI_LLOCK)
2970                 return (fs_frlock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
2971         else
2972                 return (ENOSYS);
2973 }
2974
2975 /*
2976  * Free storage space associated with the specified vnode.  The portion
2977  * to be freed is specified by bfp->l_start and bfp->l_len (already
2978  * normalized to a "whence" of 0).
2979  *
2980  * Called by fcntl(fd, F_FREESP, lkp) for libc:ftruncate, etc.
2981  */
2982 /* ARGSUSED */
2983 static int
2984 smbfs_space(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
2985         offset_t offset, cred_t *cr, caller_context_t *ct)
2986 {
2987         int             error;
2988         smbmntinfo_t    *smi;
2989
2990         smi = VTOSMI(vp);
2991
2992         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
2993                 return (EIO);
2994
2995         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
2996                 return (EIO);
2997
2998         /* Caller (fcntl) has checked v_type */
2999         ASSERT(vp->v_type == VREG);
3000         if (cmd != F_FREESP)
3001                 return (EINVAL);
3002
3003         /*
3004          * Like NFS3, no 32-bit offset checks here.
3005          * Our SMB layer takes care to return EFBIG
3006          * when it has to fallback to a 32-bit call.
3007          */
3008
3009         error = convoff(vp, bfp, 0, offset);
3010         if (!error) {
```



```
3011                 ASSERT(bfp->l_start >= 0);
3012                 if (bfp->l_len == 0) {
3013                         struct vattr va;
3014
3015                         /*
3016                          * ftruncate should not change the ctime and
3017                          * mtime if we truncate the file to its
3018                          * previous size.
3019                          */
3020                         va.va_mask = AT_SIZE;
3021                         error = smbfsgetattr(vp, &va, cr);
3022                         if (error || va.va_size == bfp->l_start)
3023                                 return (error);
3024                         va.va_mask = AT_SIZE;
3025                         va.va_size = bfp->l_start;
3026                         error = smbfssetattr(vp, &va, 0, cr);
3027                 } else
3028                         error = EINVAL;
3029         }
3030
3031         return (error);
3032 }
3033
3034 /* ARGSUSED */
3035 static int
3036 smbfs_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
3037         caller_context_t *ct)
3038 {
3039         vfs_t *vfs;
3040         smbmntinfo_t *smi;
3041         struct smb_share *ssp;
3042
3043         vfs = vp->v_vfsp;
3044         smi = VFTOSMI(vfs);
3045
3046         if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3047                 return (EIO);
3048
3049         if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3050                 return (EIO);
3051
3052         switch (cmd) {
3053         case _PC_FILESIZEBITS:
3054                 ssp = smi->smi_share;
3055                 if (SSTOVC(ssp)->vc_sopt.sv_caps & SMB_CAP_LARGE_FILES)
3056                         *valp = 64;
3057                 else
3058                         *valp = 32;
3059                 break;
3060
3061         case _PC_LINK_MAX:
3062                 /* We only ever report one link to an object */
3063                 *valp = 1;
3064                 break;
3065
3066         case _PC_ACL_ENABLED:
3067                 /*
3068                  * Always indicate that ACLs are enabled and
3069                  * that we support ACE_T format, otherwise
3070                  * libsec will ask for ACLENT_T format data
3071                  * which we don't support.
3072                  */
3073                 *valp = _ACL_ACE_ENABLED;
3074                 break;
3075
3076         case _PC_SYMLINK_MAX:   /* No symlinks until we do Unix extensions */
```

```
3077                *valp = 0;
3078                break;

3080        case _PC_XATTR_EXISTS:
3081                if (vfs->vfs_flag & VFS_XATTR) {
3082                        *valp = smbfs_xa_exists(vp, cr);
3083                        break;
3084                }
3085                return (EINVAL);

3087        case _PC_TIMESTAMP_RESOLUTION:
3088                /*
3089                 * Windows times are tenths of microseconds
3090                 * (multiples of 100 nanoseconds).
3091                 */
3092                *valp = 100L;
3093                break;

3095        default:
3096                return (fs_pathconf(vp, cmd, valp, cr, ct));
3097        }
3098        return (0);
3099 }

3101 /* ARGSUSED */
3102 static int
3103 smbfs_getsecattr(vnode_t *vp, vsecattr_t *vsa, int flag, cred_t *cr,
3104        caller_context_t *ct)
3105 {
3106        vfs_t *vfsp;
3107        smbmntinfo_t *smi;
3108        int     error;
3109        uint_t  mask;

3111        vfsp = vp->v_vfsp;
3112        smi = VFTOSMI(vfsp);

3114        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3115                return (EIO);

3117        if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3118                return (EIO);

3120        /*
3121         * Our _pathconf indicates _ACL_ACE_ENABLED,
3122         * so we should only see VSA_ACE, etc here.
3123         * Note: vn_create asks for VSA_DFACLCNT,
3124         * and it expects ENOSYS and empty data.
3125         */
3126        mask = vsa->vsa_mask & (VSA_ACE | VSA_ACECNT |
3127            VSA_ACE_ACLFLAGS | VSA_ACE_ALLTYPES);
3128        if (mask == 0)
3129                return (ENOSYS);

3131        if (smi->smi_flags & SMI_ACL)
3132                error = smbfs_acl_getvsa(vp, vsa, flag, cr);
3133        else
3134                error = ENOSYS;

3136        if (error == ENOSYS)
3137                error = fs_fab_acl(vp, vsa, flag, cr, ct);

3139        return (error);
3140 }

3142 /* ARGSUSED */
```



```
3143 static int
3144 smbfs_setsecattr(vnode_t *vp, vsecattr_t *vsa, int flag, cred_t *cr,
3145        caller_context_t *ct)
3146 {
3147        vfs_t *vfsp;
3148        smbmntinfo_t *smi;
3149        int     error;
3150        uint_t  mask;

3152        vfsp = vp->v_vfsp;
3153        smi = VFTOSMI(vfsp);

3155        if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3156                return (EIO);

3158        if (smi->smi_flags & SMI_DEAD || vfsp->vfs_flag & VFS_UNMOUNTED)
3159                return (EIO);

3161        /*
3162         * Our _pathconf indicates _ACL_ACE_ENABLED,
3163         * so we should only see VSA_ACE, etc here.
3164         */
3165        mask = vsa->vsa_mask & (VSA_ACE | VSA_ACECNT);
3166        if (mask == 0)
3167                return (ENOSYS);

3169        if (vfsp->vfs_flag & VFS_RDONLY)
3170                return (EROFS);

3172        /*
3173         * Allow only the mount owner to do this.
3174         * See comments at smbfs_access_rwx.
3175         */
3176        error = secpolicy_vnode_setdac(cr, smi->smi_uid);
3177        if (error != 0)
3178                return (error);

3180        if (smi->smi_flags & SMI_ACL)
3181                error = smbfs_acl_setvsa(vp, vsa, flag, cr);
3182        else
3183                error = ENOSYS;

3185        return (error);
3186 }


3189 /*
3190  * XXX
3191  * This op should eventually support PSARC 2007/268.
3192  */
3193 static int
3194 smbfs_shrlock(vnode_t *vp, int cmd, struct shrlock *shr, int flag, cred_t *cr,
3195        caller_context_t *ct)
3196 {
3197        if (curproc->p_zone != VTOSMI(vp)->smi_zone_ref.zref_zone)
3198                return (EIO);

3200        if (VTOSMI(vp)->smi_flags & SMI_LLOCK)
3201                return (fs_shrlock(vp, cmd, shr, flag, cr, ct));
3202        else
3203                return (ENOSYS);
3204 }

3206 static int uio_page_mapin(uio_t *uiop, page_t *pp) {
3207    u_offset_t off;
3208    size_t size;
```

```
3209     pgcnt_t npages;
3210     caddr_t kaddr;
3211     pfn_t pfnum;

3213     off = (uintptr_t) uiop->uio_loffset & PAGEOFFSET;
3214     size = P2ROUNDUP(uiop->uio_resid + off, PAGESIZE);
3215     npages = btop(size);

3217     ASSERT(pp != NULL);

3219     if (npages == 1 && kpm_enable) {
3220         kaddr = hat_kpm_mapin(pp, NULL);
3221         if (kaddr == NULL)
3222             return (EFAULT);

3224         uiop->uio_iov->iov_base = kaddr + off;
3225         uiop->uio_iov->iov_len = PAGESIZE - off;

3227     } else {
3228         kaddr = vmem_xalloc(heap_arena, size, PAGESIZE, 0, 0, NULL, NULL, VM_SLE
3229         if (kaddr == NULL)
3230             return (EFAULT);

3232         uiop->uio_iov->iov_base = kaddr + off;
3233         uiop->uio_iov->iov_len = size - off;

3235         /*map pages into kaddr*/
3236         uint_t attr = PROT_READ | PROT_WRITE | HAT_NOSYNC;
3237         while (npages-- > 0) {
3238             pfnum = pp->p_pagenum;
3239             pp = pp->p_next;

3241             hat_devload(kas.a_hat, kaddr, PAGESIZE, pfnum, attr, HAT_LOAD_LOCK);
3242             kaddr += PAGESIZE;
3243         }
3244     }
3245     return (0);
3246 }

3248 static void uio_page_mapout(uio_t *uiop, page_t *pp) {
3249     u_offset_t off;
3250     size_t size;
3251     pgcnt_t npages;
3252     caddr_t kaddr;

3254     kaddr = uiop->uio_iov->iov_base;
3255     off = (uintptr_t) kaddr & PAGEOFFSET;
3256     size = P2ROUNDUP(uiop->uio_iov->iov_len + off, PAGESIZE);
3257     npages = btop(size);

3259     ASSERT(pp != NULL);

3261     kaddr = (caddr_t) ((uintptr_t) kaddr & MMU_PAGEMASK);

3263     if (npages == 1 && kpm_enable) {
3264         hat_kpm_mapout(pp, NULL, kaddr);

3266     } else {
3267         hat_unload(kas.a_hat, (void*) kaddr, size,
3268                 HAT_UNLOAD_NOSYNC | HAT_UNLOAD_UNLOCK);
3269         vmem_free(heap_arena, (void*) kaddr, size);
3270     }
3271     uiop->uio_iov->iov_base = 0;
3272     uiop->uio_iov->iov_len = 0;
3273 }
```



```
3275 static int smbfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
3276         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
3277         caller_context_t *ct) {
3278     smbnode_t *np;
3279     smbmntinfo_t *smi;
3280     struct vattr va;
3281     segvn_crargs_t vn_a;
3282     int error;

3284     np = VTOSMB(vp);
3285     smi = VTOSMI(vp);

3287     if (curproc->p_zone != smi->smi_zone_ref.zref_zone)
3288         return (EIO);

3290     if (smi->smi_flags & SMI_DEAD || vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
3291         return (EIO);

3293     if (vp->v_flag & VNOMAP || vp->v_flag & VNOCACHE)
3294         return (EAGAIN);

3296     if (vp->v_type != VREG)
3297         return (ENODEV);

3299     va.va_mask = AT_ALL;
3300     if (error = smbfsgetattr(vp, &va, cr))
3301         return (error);

3303     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_WRITER, SMBINTR(vp)))
3304         return (EINTR);

3306     if (MANDLOCK(vp, va.va_mode)) {
3307         error = EAGAIN;
3308         goto out;
3309     }

3311     as_rangelock(as);
3312     error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);

3314     if (error != 0) {
3315         as_rangeunlock(as);
3316         goto out;
3317     }

3319     vn_a.vp = vp;
3320     vn_a.offset = off;
3321     vn_a.type = flags & MAP_TYPE;
3322     vn_a.prot = prot;
3323     vn_a.maxprot = maxprot;
3324     vn_a.flags = flags & ~MAP_TYPE;
3325     vn_a.cred = cr;
3326     vn_a.amp = NULL;
3327     vn_a.szc = 0;
3328     vn_a.lgrp_mem_policy_flags = 0;

3330     error = as_map(as, *addrp, len, segvn_create, &vn_a);

3332     as_rangeunlock(as);

3334 out:
3335     smbfs_rw_exit(&np->r_lkserlock);

3337     return (error);
3338 }

3340 static int smbfs_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
```

```
3341         size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
3342         caller_context_t *ct) {
3343     atomic_add_long((ulong_t *) & VTOSMB(vp)->r_mapcnt, btopr(len));
3344     return (0);
3345 }

3347 static int smbfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
3348         size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
3349         caller_context_t *ct) {

3351     smbnode_t *np;

3353     atomic_add_long((ulong_t *) & VTOSMB(vp)->r_mapcnt, -btopr(len));

3355     /* mark RDIRTY here, will be used to check if a file is dirty when unmount s
3356     if (vn_has_cached_data(vp) && !vn_is_readonly(vp) && maxprot & PROT_WRITE &&
3357         np = VTOSMB(vp);
3358         mutex_enter(&np->r_statelock);
3359         np->r_flags |= RDIRTY;
3360         mutex_exit(&np->r_statelock);
3361     }
3362     return (0);
3363 }

3365 static int smbfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags,
3366         cred_t *cr, caller_context_t *ct) {

3368     smbnode_t *np;
3369     size_t io_len;
3370     u_offset_t io_off;
3371     u_offset_t eoff;
3372     int error = 0;
3373     page_t *pp;

3375     np = VTOSMB(vp);

3377     if (len == 0) {
3378         /* will flush the file, so clear RDIRTY */
3379         if (off == (u_offset_t) 0 && (np->r_flags & RDIRTY)) {
3380             mutex_enter(&np->r_statelock);
3381             np->r_flags &= ~RDIRTY;
3382             mutex_exit(&np->r_statelock);
3383         }

3385         error = pvn_vplist_dirty(vp, off, smbfs_putapage, flags, cr);
3386     } else {

3388         eoff = off + len;

3390         mutex_enter(&np->r_statelock);
3391         if (eoff > np->r_size)
3392             eoff = np->r_size;
3393         mutex_exit(&np->r_statelock);

3395         for (io_off = off; io_off < eoff; io_off += io_len) {
3396             if ((flags & B_INVAL) || (flags & B_ASYNC) == 0) {
3397                 pp = page_lookup(vp, io_off,
3398                         (flags & (B_INVAL | B_FREE) ? SE_EXCL : SE_SHARED));
3399             } else {
3400                 pp = page_lookup_nowait(vp, io_off,
3401                         (flags & B_FREE) ? SE_EXCL : SE_SHARED);
3402             }

3404             if (pp == NULL || !pvn_getdirty(pp, flags))
3405                 io_len = PAGESIZE;
3406             else {
```



```
3407                 error = smbfs_putapage(vp, pp, &io_off, &io_len, flags, cr);
3408             }
3409         }

3411     }

3413     return (error);
3414 }

3416 static int smbfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *len
3417         int flags, cred_t *cr) {

3419     struct smb_cred scred;
3420     smbnode_t *np;
3421     smbmntinfo_t *smi;
3422     smb_share_t *ssp;
3423     uio_t uio;
3424     iovec_t uiov, uiov_bak;

3426     size_t io_len;
3427     u_offset_t io_off;
3428     size_t bsize;
3429     size_t blksize;
3430     u_offset_t blkoff;
3431     int error;

3433     np = VTOSMB(vp);
3434     smi = VTOSMI(vp);
3435     ssp = smi->smi_share;

3437     /*do block io, get a kluster of dirty pages in a block.*/
3438     bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
3439     blkoff = pp->p_offset / bsize;
3440     blkoff *= bsize;
3441     blksize = roundup(bsize, PAGESIZE);

3443     pp = pvn_write_kluster(vp, pp, &io_off, &io_len, blkoff, blksize, flags);

3445     ASSERT(pp->p_offset >= blkoff);

3447     if (io_off + io_len > blkoff + blksize) {
3448         ASSERT((io_off + io_len)-(blkoff + blksize) < PAGESIZE);
3449         io_len = blkoff + blksize - io_off;
3450     }

3452     /*currently, don't allow put pages beyond EOF, unless smbfs_read/smbfs_write
3453      *can do io through segkpm or vpm.*/
3454     mutex_enter(&np->r_statelock);
3455     if (io_off >= np->r_size) {
3456         mutex_exit(&np->r_statelock);
3457         error = 0;
3458         goto out;
3459     } else if (io_off + io_len > np->r_size) {
3460         int npages = btopr(np->r_size - io_off);
3461         page_t *trunc;
3462         page_list_break(&pp, &trunc, npages);
3463         if (trunc)
3464             pvn_write_done(trunc, flags);
3465         io_len = np->r_size - io_off;
3466     }
3467     mutex_exit(&np->r_statelock);

3469     if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
3470         return (EINTR);
3471     smb_credinit(&scred, cr);
```

```
3473     if (np->n_vcgenid != ssp->ss_vcgenid)
3474         error = ESTALE;
3475     else {
3476         /*just use uio instead of buf, since smb_rwuio need uio.*/
3477         uiov.iov_base = 0;
3478         uiov.iov_len = 0;
3479         uio.uio_iov = &uiov;
3480         uio.uio_iovcnt = 1;
3481         uio.uio_loffset = io_off;
3482         uio.uio_resid = io_len;
3483         uio.uio_segflg = UIO_SYSSPACE;
3484         uio.uio_llimit = MAXOFFSET_T;
3485         /*map pages into kernel address space, and setup uio.*/
3486         error = uio_page_mapin(&uio, pp);
3487         if (error == 0) {
3488             uiov_bak.iov_base = uiov.iov_base;
3489             uiov_bak.iov_len = uiov.iov_len;
3490             error = smb_rwuio(ssp, np->n_fid, UIO_WRITE, &uio, &scred, smb_timo_
3491             if (error == 0) {
3492                 mutex_enter(&np->r_statelock);
3493                 np->n_flag |= (NFLUSHWIRE | NATTRCHANGED);
3494                 mutex_exit(&np->r_statelock);
3495                 (void) smbfs_smb_flush(np, &scred);
3496             }
3497             /*unmap pages from kernel address space.*/
3498             uio.uio_iov = &uiov_bak;
3499             uio_page_mapout(&uio, pp);
3500         }
3501     }

3503     smb_credrele(&scred);
3504     smbfs_rw_exit(&np->r_lkserlock);

3506 out:
3507     pvn_write_done(pp, ((error) ? B_ERROR : 0) | B_WRITE | flags);

3509     if (offp)
3510         *offp = io_off;
3511     if (lenp)
3512         *lenp = io_len;

3514     return (error);
3515 }

3517 static int smbfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
3518         page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3519         enum seg_rw rw, cred_t *cr, caller_context_t *ct) {

3521     int error;

3523     /*these pages have all protections.*/
3524     if (protp)
3525         *protp = PROT_ALL;

3527     if (len <= PAGESIZE) {
3528         error = smbfs_getapage(vp, off, len, protp, pl, plsz, seg, addr, rw,
3529                 cr);
3530     } else {
3531         error = pvn_getpages(smbfs_getapage, vp, off, len, protp, pl, plsz, seg,
3532                 addr, rw, cr);
3533     }

3535     return (error);
3536 }

3538 static int smbfs_getapage(vnode_t *vp, u_offset_t off, size_t len,
```



```
3539         uint_t *protp, page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3540         enum seg_rw rw, cred_t *cr) {

3542     smbnode_t *np;
3543     smbmntinfo_t *smi;
3544     smb_share_t *ssp;
3545     smb_cred_t scred;

3547     page_t *pp;
3548     uio_t uio;
3549     iovec_t uiov, uiov_bak;

3551     u_offset_t blkoff;
3552     size_t bsize;
3553     size_t blksize;

3555     u_offset_t io_off;
3556     size_t io_len;
3557     size_t pages_len;

3559     int error = 0;

3561     np = VTOSMB(vp);
3562     smi = VTOSMI(vp);
3563     ssp = smi->smi_share;

3565     /*if pl is null,it's meaningless*/
3566     if (pl == NULL)
3567         return (EFAULT);

3569 again:
3570     if (page_exists(vp, off) == NULL) {
3571         if (rw == S_CREATE) {
3572             /*just return a empty page if asked to create.*/
3573             if ((pp = page_create_va(vp, off, PAGESIZE, PG_WAIT | PG_EXCL, seg,
3574                 goto again;
3575             pages_len = PAGESIZE;
3576         } else {

3578             /*do block io, get a kluster of non-exist pages in a block.*/
3579             bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
3580             blkoff = off / bsize;
3581             blkoff *= bsize;
3582             blksize = roundup(bsize, PAGESIZE);

3584             pp = pvn_read_kluster(vp, off, seg, addr, &io_off, &io_len, blkoff,

3586             if (pp == NULL)
3587                 goto again;

3589             pages_len = io_len;

3591             /*currently, don't allow get pages beyond EOF, unless smbfs_read/smb
3592              *can do io through segkpm or vpm.*/
3593             mutex_enter(&np->r_statelock);
3594             if (io_off >= np->r_size) {
3595                 mutex_exit(&np->r_statelock);
3596                 error = 0;
3597                 goto out;
3598             } else if (io_off + io_len > np->r_size) {
3599                 int npages = btopr(np->r_size - io_off);
3600                 page_t *trunc;

3602                 page_list_break(&pp, &trunc, npages);
3603                 if (trunc)
3604                     pvn_read_done(trunc, 0);
```

```
3605                 io_len = np->r_size - io_off;
3606             }
3607             mutex_exit(&np->r_statelock);

3609             if (smbfs_rw_enter_sig(&np->r_lkserlock, RW_READER, SMBINTR(vp)))
3610                 return EINTR;
3611             smb_credinit(&scred, cr);

3613             /*just use uio instead of buf, since smb_rwuio need uio.*/
3614             uiov.iov_base = 0;
3615             uiov.iov_len = 0;
3616             uio.uio_iov = &uiov;
3617             uio.uio_iovcnt = 1;
3618             uio.uio_loffset = io_off;
3619             uio.uio_resid = io_len;
3620             uio.uio_segflg = UIO_SYSSPACE;
3621             uio.uio_llimit = MAXOFFSET_T;

3623             /*map pages into kernel address space, and setup uio.*/
3624             error = uio_page_mapin(&uio, pp);
3625             if (error == 0) {
3626                 uiov_bak.iov_base = uiov.iov_base;
3627                 uiov_bak.iov_len = uiov.iov_len;
3628                 error = smb_rwuio(ssp, np->n_fid, UIO_READ, &uio, &scred, smb_ti
3629                 /*unmap pages from kernel address space.*/
3630                 uio.uio_iov = &uiov_bak;
3631                 uio_page_mapout(&uio, pp);
3632             }

3634             smb_credrele(&scred);
3635             smbfs_rw_exit(&np->r_lkserlock);
3636         }
3637     } else {
3638         se_t se = rw == S_CREATE ? SE_EXCL : SE_SHARED;
3639         if ((pp = page_lookup(vp, off, se)) == NULL) {
3640             goto again;
3641         }
3642     }

3644 out:
3645     if (pp) {
3646         if (error) {
3647             pvn_read_done(pp, B_ERROR);
3648         } else {
3649             /*init page list, unlock pages.*/
3650             pvn_plist_init(pp, pl, plsz, off, pages_len, rw);
3651         }
3652     }

3654     return (error);
3655 }


3658 void smbfs_invalidate_pages(vnode_t *vp, u_offset_t off, cred_t *cr) {

3660     smbnode_t *np;

3662     np = VTOSMB(vp);
3663     /* will flush the file, so clear RDIRTY */
3664     if (off == (u_offset_t) 0 && (np->r_flags & RDIRTY)) {
3665         mutex_enter(&np->r_statelock);
3666         np->r_flags &= ~RDIRTY;
3667         mutex_exit(&np->r_statelock);
3668     }

3670     (void) pvn_vplist_dirty(vp, off, smbfs_putapage, B_INVAL | B_TRUNC, cr);
```



```
3671 }


3674 #endif /* ! codereview */
```