```
**********************************************************
  254228 Thu Jan  8 09:14:33 2015
new/usr/src/uts/common/fs/cachefs/cachefs_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1992, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/systm.h>
#include <sys/cred.h>
#include <sys/proc.h>
#include <sys/user.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/file.h>
#include <sys/filio.h>
#include <sys/uio.h>
#include <sys/buf.h>
#include <sys/mman.h>
#include <sys/tiuser.h>
#include <sys/pathname.h>
#include <sys/dirent.h>
#include <sys/conf.h>
#include <sys/debug.h>
#include <sys/vmsystm.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/swap.h>
#include <sys/errno.h>
#include <sys/sysmacros.h>
#include <sys/disp.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/vtrace.h>
#include <sys/mount.h>
#include <sys/bootconf.h>
#include <sys/dnlc.h>
#include <sys/stat.h>
#include <sys/acl.h>
```



```
#include <sys/policy.h>
#include <rpc/types.h>

#include <vm/hat.h>
#include <vm/as.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/seg.h>
#include <vm/seg_map.h>
#include <vm/seg_vn.h>
#include <vm/rm.h>
#include <sys/fs/cachefs_fs.h>
#include <sys/fs/cachefs_dir.h>
#include <sys/fs/cachefs_dlog.h>
#include <sys/fs/cachefs_ioctl.h>
#include <sys/fs/cachefs_log.h>
#include <fs/fs_subr.h>

int cachefs_dnlc;       /* use dnlc, debugging */

static void cachefs_attr_setup(vattr_t *srcp, vattr_t *targp, cnode_t *cp,
    cred_t *cr);
static void cachefs_creategid(cnode_t *dcp, cnode_t *newcp, vattr_t *vap,
    cred_t *cr);
static void cachefs_createacl(cnode_t *dcp, cnode_t *newcp);
static int cachefs_getaclfromcache(cnode_t *cp, vsecattr_t *vsec);
static int cachefs_getacldirvp(cnode_t *cp);
static void cachefs_acl2perm(cnode_t *cp, vsecattr_t *vsec);
static int cachefs_access_local(void *cp, int mode, cred_t *cr);
static int cachefs_acl_access(struct cnode *cp, int mode, cred_t *cr);
static int cachefs_push_connected(vnode_t *vp, struct buf *bp, size_t iolen,
    u_offset_t iooff, cred_t *cr);
static int cachefs_push_front(vnode_t *vp, struct buf *bp, size_t iolen,
    u_offset_t iooff, cred_t *cr);
static int cachefs_setattr_connected(vnode_t *vp, vattr_t *vap, int flags,
    cred_t *cr, caller_context_t *ct);
static int cachefs_setattr_disconnected(vnode_t *vp, vattr_t *vap,
    int flags, cred_t *cr, caller_context_t *ct);
static int cachefs_access_connected(struct vnode *vp, int mode,
    int flags, cred_t *cr);
static int cachefs_lookup_back(vnode_t *dvp, char *nm, vnode_t **vpp,
    cred_t *cr);
static int cachefs_symlink_connected(vnode_t *dvp, char *lnm, vattr_t *tva,
    char *tnm, cred_t *cr);
static int cachefs_symlink_disconnected(vnode_t *dvp, char *lnm,
    vattr_t *tva, char *tnm, cred_t *cr);
static int cachefs_link_connected(vnode_t *tdvp, vnode_t *fvp, char *tnm,
    cred_t *cr);
static int cachefs_link_disconnected(vnode_t *tdvp, vnode_t *fvp,
    char *tnm, cred_t *cr);
static int cachefs_mkdir_connected(vnode_t *dvp, char *nm, vattr_t *vap,
    vnode_t **vpp, cred_t *cr);
static int cachefs_mkdir_disconnected(vnode_t *dvp, char *nm, vattr_t *vap,
    vnode_t **vpp, cred_t *cr);
static int cachefs_stickyrmchk(struct cnode *dcp, struct cnode *cp, cred_t *cr);
static int cachefs_rmdir_connected(vnode_t *dvp, char *nm,
    vnode_t *cdir, cred_t *cr, vnode_t *vp);
static int cachefs_rmdir_disconnected(vnode_t *dvp, char *nm,
    vnode_t *cdir, cred_t *cr, vnode_t *vp);
static char *cachefs_newname(void);
static int cachefs_remove_dolink(vnode_t *dvp, vnode_t *vp, char *nm,
    cred_t *cr);
static int cachefs_rename_connected(vnode_t *odvp, char *onm,
    vnode_t *ndvp, char *nnm, cred_t *cr, vnode_t *delvp);
static int cachefs_rename_disconnected(vnode_t *odvp, char *onm,
    vnode_t *ndvp, char *nnm, cred_t *cr, vnode_t *delvp);
```

```
128 static int cachefs_readdir_connected(vnode_t *vp, uio_t *uiop, cred_t *cr,
129     int *eofp);
130 static int cachefs_readdir_disconnected(vnode_t *vp, uio_t *uiop,
131     cred_t *cr, int *eofp);
132 static int cachefs_readback_translate(cnode_t *cp, uio_t *uiop,
133         cred_t *cr, int *eofp);

135 static int cachefs_setattr_common(vnode_t *vp, vattr_t *vap, int flags,
136     cred_t *cr, caller_context_t *ct);

138 static  int     cachefs_open(struct vnode **, int, cred_t *,
139                         caller_context_t *);
140 static  int     cachefs_close(struct vnode *, int, int, offset_t,
141                         cred_t *, caller_context_t *);
142 static  int     cachefs_read(struct vnode *, struct uio *, int, cred_t *,
143                         caller_context_t *);
144 static  int     cachefs_write(struct vnode *, struct uio *, int, cred_t *,
145                         caller_context_t *);
146 static  int     cachefs_ioctl(struct vnode *, int, intptr_t, int, cred_t *,
147                         int *, caller_context_t *);
148 static  int     cachefs_getattr(struct vnode *, struct vattr *, int,
149                         cred_t *, caller_context_t *);
150 static  int     cachefs_setattr(struct vnode *, struct vattr *,
151                         int, cred_t *, caller_context_t *);
152 static  int     cachefs_access(struct vnode *, int, int, cred_t *,
153                         caller_context_t *);
154 static  int     cachefs_lookup(struct vnode *, char *, struct vnode **,
155                         struct pathname *, int, struct vnode *, cred_t *,
156                         caller_context_t *, int *, pathname_t *);
157 static  int     cachefs_create(struct vnode *, char *, struct vattr *,
158                         enum vcexcl, int, struct vnode **, cred_t *, int,
159                         caller_context_t *, vsecattr_t *);
160 static  int     cachefs_create_connected(vnode_t *dvp, char *nm,
161                         vattr_t *vap, enum vcexcl exclusive, int mode,
162                         vnode_t **vpp, cred_t *cr);
163 static  int     cachefs_create_disconnected(vnode_t *dvp, char *nm,
164                         vattr_t *vap, enum vcexcl exclusive, int mode,
165                         vnode_t **vpp, cred_t *cr);
166 static  int     cachefs_remove(struct vnode *, char *, cred_t *,
167                         caller_context_t *, int);
168 static  int     cachefs_link(struct vnode *, struct vnode *, char *,
169                         cred_t *, caller_context_t *, int);
170 static  int     cachefs_rename(struct vnode *, char *, struct vnode *,
171                         char *, cred_t *, caller_context_t *, int);
172 static  int     cachefs_mkdir(struct vnode *, char *, struct
173                         vattr *, struct vnode **, cred_t *, caller_context_t *,
174                         int, vsecattr_t *);
175 static  int     cachefs_rmdir(struct vnode *, char *, struct vnode *,
176                         cred_t *, caller_context_t *, int);
177 static  int     cachefs_readdir(struct vnode *, struct uio *,
178                         cred_t *, int *, caller_context_t *, int);
179 static  int     cachefs_symlink(struct vnode *, char *, struct vattr *,
180                         char *, cred_t *, caller_context_t *, int);
181 static  int     cachefs_readlink(struct vnode *, struct uio *, cred_t *,
182                         caller_context_t *);
183 static int cachefs_readlink_connected(vnode_t *vp, uio_t *uiop, cred_t *cr);
184 static int cachefs_readlink_disconnected(vnode_t *vp, uio_t *uiop);
185 static  int     cachefs_fsync(struct vnode *, int, cred_t *,
186                         caller_context_t *);
187 static  void    cachefs_inactive(struct vnode *, cred_t *, caller_context_t *);
188 static  int     cachefs_fid(struct vnode *, struct fid *, caller_context_t *);
189 static  int     cachefs_rwlock(struct vnode *, int, caller_context_t *);
190 static  void    cachefs_rwunlock(struct vnode *, int, caller_context_t *);
191 static  int     cachefs_seek(struct vnode *, offset_t, offset_t *,
192                         caller_context_t *);
193 static  int     cachefs_frlock(struct vnode *, int, struct flock64 *,
```



```
194                         int, offset_t, struct flk_callback *, cred_t *,
195                         caller_context_t *);
196 static  int     cachefs_space(struct vnode *, int, struct flock64 *, int,
197                         offset_t, cred_t *, caller_context_t *);
198 static  int     cachefs_realvp(struct vnode *, struct vnode **,
199                         caller_context_t *);
200 static  int     cachefs_getpage(struct vnode *, offset_t, size_t, uint_t *,
201                         struct page *[], size_t, struct seg *, caddr_t,
202                         enum seg_rw, cred_t *, caller_context_t *);
203 static  int     cachefs_getapage(struct vnode *, u_offset_t, size_t, uint_t *,
204                         struct page *[], size_t, struct seg *, caddr_t,
205                         enum seg_rw, cred_t *);
206 static  int     cachefs_getapage_back(struct vnode *, u_offset_t, size_t,
207                         uint_t *, struct page *[], size_t, struct seg *, caddr_t,
208                         enum seg_rw, cred_t *);
209 static  int     cachefs_putpage(struct vnode *, offset_t, size_t, int,
210                         cred_t *, caller_context_t *);
211 static  int     cachefs_map(struct vnode *, offset_t, struct as *,
212                         caddr_t *, size_t, uchar_t, uchar_t, uint_t, cred_t *,
213                         caller_context_t *);
214 static  int     cachefs_addmap(struct vnode *, offset_t, struct as *,
215                         caddr_t, size_t, uchar_t, uchar_t, uint_t, cred_t *,
216                         caller_context_t *);
217 static  int     cachefs_delmap(struct vnode *, offset_t, struct as *,
218                         caddr_t, size_t, uint_t, uint_t, uint_t, cred_t *,
219                         caller_context_t *);
220 static int      cachefs_setsecattr(vnode_t *vp, vsecattr_t *vsec,
221                         int flag, cred_t *cr, caller_context_t *);
222 static int      cachefs_getsecattr(vnode_t *vp, vsecattr_t *vsec,
223                         int flag, cred_t *cr, caller_context_t *);
224 static  int     cachefs_shrlock(vnode_t *, int, struct shrlock *, int,
225                         cred_t *, caller_context_t *);
226 static int cachefs_getsecattr_connected(vnode_t *vp, vsecattr_t *vsec, int flag,
227     cred_t *cr);
228 static int cachefs_getsecattr_disconnected(vnode_t *vp, vsecattr_t *vsec,
229     int flag, cred_t *cr);

231 static int      cachefs_dump(struct vnode *, caddr_t, offset_t, offset_t,
232                         caller_context_t *);
233 static int      cachefs_pageio(struct vnode *, page_t *,
234                     u_offset_t, size_t, int, cred_t *, caller_context_t *);
235 static int      cachefs_writepage(struct vnode *vp, caddr_t base,
236                     int tcount, struct uio *uiop);
237 static int      cachefs_pathconf(vnode_t *, int, ulong_t *, cred_t *,
238                         caller_context_t *);

240 static int      cachefs_read_backfs_nfsv4(vnode_t *vp, uio_t *uiop, int ioflag,
241                         cred_t *cr, caller_context_t *ct);
242 static int      cachefs_write_backfs_nfsv4(vnode_t *vp, uio_t *uiop, int ioflag,
243                         cred_t *cr, caller_context_t *ct);
244 static int      cachefs_getattr_backfs_nfsv4(vnode_t *vp, vattr_t *vap,
245                         int flags, cred_t *cr, caller_context_t *ct);
246 static int      cachefs_remove_backfs_nfsv4(vnode_t *dvp, char *nm, cred_t *cr,
247                         vnode_t *vp);
248 static int      cachefs_getpage_backfs_nfsv4(struct vnode *vp, offset_t off,
249                         size_t len, uint_t *protp, struct page *pl[],
250                         size_t plsz, struct seg *seg, caddr_t addr,
251                         enum seg_rw rw, cred_t *cr);
252 static int      cachefs_putpage_backfs_nfsv4(vnode_t *vp, offset_t off,
253                         size_t len, int flags, cred_t *cr);
254 static int      cachefs_map_backfs_nfsv4(struct vnode *vp, offset_t off,
255                         struct as *as, caddr_t *addrp, size_t len, uchar_t prot,
256                         uchar_t maxprot, uint_t flags, cred_t *cr);
257 static int      cachefs_space_backfs_nfsv4(struct vnode *vp, int cmd,
258                         struct flock64 *bfp, int flag, offset_t offset,
259                         cred_t *cr, caller_context_t *ct);
```

```
261 struct vnodeops *cachefs_vnodeops;

263 static const fs_operation_def_t cachefs_vnodeops_template[] = {
264         VOPNAME_OPEN,           { .vop_open = cachefs_open },
265         VOPNAME_CLOSE,          { .vop_close = cachefs_close },
266         VOPNAME_READ,           { .vop_read = cachefs_read },
267         VOPNAME_WRITE,          { .vop_write = cachefs_write },
268         VOPNAME_IOCTL,          { .vop_ioctl = cachefs_ioctl },
269         VOPNAME_GETATTR,        { .vop_getattr = cachefs_getattr },
270         VOPNAME_SETATTR,        { .vop_setattr = cachefs_setattr },
271         VOPNAME_ACCESS,         { .vop_access = cachefs_access },
272         VOPNAME_LOOKUP,         { .vop_lookup = cachefs_lookup },
273         VOPNAME_CREATE,         { .vop_create = cachefs_create },
274         VOPNAME_REMOVE,         { .vop_remove = cachefs_remove },
275         VOPNAME_LINK,           { .vop_link = cachefs_link },
276         VOPNAME_RENAME,         { .vop_rename = cachefs_rename },
277         VOPNAME_MKDIR,          { .vop_mkdir = cachefs_mkdir },
278         VOPNAME_RMDIR,          { .vop_rmdir = cachefs_rmdir },
279         VOPNAME_READDIR,        { .vop_readdir = cachefs_readdir },
280         VOPNAME_SYMLINK,        { .vop_symlink = cachefs_symlink },
281         VOPNAME_READLINK,       { .vop_readlink = cachefs_readlink },
282         VOPNAME_FSYNC,          { .vop_fsync = cachefs_fsync },
283         VOPNAME_INACTIVE,       { .vop_inactive = cachefs_inactive },
284         VOPNAME_FID,            { .vop_fid = cachefs_fid },
285         VOPNAME_RWLOCK,         { .vop_rwlock = cachefs_rwlock },
286         VOPNAME_RWUNLOCK,       { .vop_rwunlock = cachefs_rwunlock },
287         VOPNAME_SEEK,           { .vop_seek = cachefs_seek },
288         VOPNAME_FRLOCK,         { .vop_frlock = cachefs_frlock },
289         VOPNAME_SPACE,          { .vop_space = cachefs_space },
290         VOPNAME_REALVP,         { .vop_realvp = cachefs_realvp },
291         VOPNAME_GETPAGE,        { .vop_getpage = cachefs_getpage },
292         VOPNAME_PUTPAGE,        { .vop_putpage = cachefs_putpage },
293         VOPNAME_MAP,            { .vop_map = cachefs_map },
294         VOPNAME_ADDMAP,         { .vop_addmap = cachefs_addmap },
295         VOPNAME_DELMAP,         { .vop_delmap = cachefs_delmap },
296         VOPNAME_DUMP,           { .vop_dump = cachefs_dump },
297         VOPNAME_PATHCONF,       { .vop_pathconf = cachefs_pathconf },
298         VOPNAME_PAGEIO,         { .vop_pageio = cachefs_pageio },
299         VOPNAME_SETSECATTR,     { .vop_setsecattr = cachefs_setsecattr },
300         VOPNAME_GETSECATTR,     { .vop_getsecattr = cachefs_getsecattr },
301         VOPNAME_SHRLOCK,        { .vop_shrlock = cachefs_shrlock },
302         NULL,                   NULL
303 };

305 /* forward declarations of statics */
306 static void cachefs_modified(cnode_t *cp);
307 static int cachefs_modified_alloc(cnode_t *cp);

309 int
310 cachefs_init_vnops(char *name)
311 {
312         return (vn_make_ops(name,
313             cachefs_vnodeops_template, &cachefs_vnodeops));
314 }

316 struct vnodeops *
317 cachefs_getvnodeops(void)
318 {
319         return (cachefs_vnodeops);
320 }

322 static int
323 cachefs_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
324 {
325         int error = 0;
```



```
326         cnode_t *cp = VTOC(*vpp);
327         fscache_t *fscp = C_TO_FSCACHE(cp);
328         int held = 0;
329         int type;
330         int connected = 0;

332 #ifdef CFSDEBUG
333         CFS_DEBUG(CFSDEBUG_VOPS)
334                 printf("cachefs_open: ENTER vpp %p flag %x\n",
335                     (void *)vpp, flag);
336 #endif
337         if (getzoneid() != GLOBAL_ZONEID) {
338                 error = EPERM;
339                 goto out;
340         }
341         if ((flag & FWRITE) &&
342             ((*vpp)->v_type == VDIR || (*vpp)->v_type == VLNK)) {
343                 error = EISDIR;
344                 goto out;
345         }

347         /*
348          * Cachefs only provides pass-through support for NFSv4,
349          * and all vnode operations are passed through to the
350          * back file system. For NFSv4 pass-through to work, only
351          * connected operation is supported, the cnode backvp must
352          * exist, and cachefs optional (eg., disconnectable) flags
353          * are turned off. Assert these conditions to ensure that
354          * the backfilesystem is called for the open operation.
355          */
356         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
357         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

359         for (;;) {
360                 /* get (or renew) access to the file system */
361                 if (held) {
362                         /* Won't loop with NFSv4 connected behavior */
363                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
364                         cachefs_cd_release(fscp);
365                         held = 0;
366                 }
367                 error = cachefs_cd_access(fscp, connected, 0);
368                 if (error)
369                         goto out;
370                 held = 1;

372                 mutex_enter(&cp->c_statelock);

374                 /* grab creds if we do not have any yet */
375                 if (cp->c_cred == NULL) {
376                         crhold(cr);
377                         cp->c_cred = cr;
378                 }
379                 cp->c_flags |= CN_NEEDOPEN;

381                 /* if we are disconnected */
382                 if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
383                         /* if we cannot write to the file system */
384                         if ((flag & FWRITE) && CFS_ISFS_WRITE_AROUND(fscp)) {
385                                 mutex_exit(&cp->c_statelock);
386                                 connected = 1;
387                                 continue;
388                         }
389                         /*
390                          * Allow read only requests to continue
391                          */
```

```
392                         if ((flag & (FWRITE|FREAD)) == FREAD) {
393                                 /* track the flag for opening the backvp */
394                                 cp->c_rdcnt++;
395                                 mutex_exit(&cp->c_statelock);
396                                 error = 0;
397                                 break;
398                         }
399 
400                         /*
401                          * check credentials  - if this procs
402                          * credentials don't match the creds in the
403                          * cnode disallow writing while disconnected.
404                          */
405                         if (crcmp(cp->c_cred, CRED()) != 0 &&
406                             secpolicy_vnode_access2(CRED(), *vpp,
407                             cp->c_attr.va_uid, 0, VWRITE) != 0) {
408                                 mutex_exit(&cp->c_statelock);
409                                 connected = 1;
410                                 continue;
411                         }
412                         /* to get here, we know that the WRITE flag is on */
413                         cp->c_wrcnt++;
414                         if (flag & FREAD)
415                                 cp->c_rdcnt++;
416                 }
417 
418                 /* else if we are connected */
419                 else {
420                         /* if cannot use the cached copy of the file */
421                         if ((flag & FWRITE) && CFS_ISFS_WRITE_AROUND(fscp) &&
422                             ((cp->c_flags & CN_NOCACHE) == 0))
423                                 cachefs_nocache(cp);
424 
425                         /* pass open to the back file */
426                         if (cp->c_backvp) {
427                                 cp->c_flags &= ~CN_NEEDOPEN;
428                                 CFS_DPRINT_BACKFS_NFSV4(fscp,
429                                     ("cachefs_open (nfsv4): cnode %p, "
430                                     "backvp %p\n", cp, cp->c_backvp));
431                                 error = VOP_OPEN(&cp->c_backvp, flag, cr, ct);
432                                 if (CFS_TIMEOUT(fscp, error)) {
433                                         mutex_exit(&cp->c_statelock);
434                                         cachefs_cd_release(fscp);
435                                         held = 0;
436                                         cachefs_cd_timedout(fscp);
437                                         continue;
438                                 } else if (error) {
439                                         mutex_exit(&cp->c_statelock);
440                                         break;
441                                 }
442                         } else {
443                                 /* backvp will be VOP_OPEN'd later */
444                                 if (flag & FREAD)
445                                         cp->c_rdcnt++;
446                                 if (flag & FWRITE)
447                                         cp->c_wrcnt++;
448                         }
449 
450                         /*
451                          * Now perform a consistency check on the file.
452                          * If strict consistency then force a check to
453                          * the backfs even if the timeout has not expired
454                          * for close-to-open consistency.
455                          */
456                         type = 0;
457                         if (fscp->fs_consttype == CFS_FS_CONST_STRICT)
```



```
458                                 type = C_BACK_CHECK;
459                         error = CFSOP_CHECK_COBJECT(fscp, cp, type, cr);
460                         if (CFS_TIMEOUT(fscp, error)) {
461                                 mutex_exit(&cp->c_statelock);
462                                 cachefs_cd_release(fscp);
463                                 held = 0;
464                                 cachefs_cd_timedout(fscp);
465                                 continue;
466                         }
467                 }
468                 mutex_exit(&cp->c_statelock);
469                 break;
470         }
471         if (held)
472                 cachefs_cd_release(fscp);
473 out:
474 #ifdef CFS_CD_DEBUG
475         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
476 #endif
477 #ifdef CFSDEBUG
478         CFS_DEBUG(CFSDEBUG_VOPS)
479                 printf("cachefs_open: EXIT vpp %p error %d\n",
480                     (void *)vpp, error);
481 #endif
482         return (error);
483 }
484 
485 /* ARGSUSED */
486 static int
487 cachefs_close(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
488         caller_context_t *ct)
489 {
490         int error = 0;
491         cnode_t *cp = VTOC(vp);
492         fscache_t *fscp = C_TO_FSCACHE(cp);
493         int held = 0;
494         int connected = 0;
495         int close_cnt = 1;
496         cachefscache_t *cachep;
497 
498 #ifdef CFSDEBUG
499         CFS_DEBUG(CFSDEBUG_VOPS)
500                 printf("cachefs_close: ENTER vp %p\n", (void *)vp);
501 #endif
502         /*
503          * Cachefs only provides pass-through support for NFSv4,
504          * and all vnode operations are passed through to the
505          * back file system. For NFSv4 pass-through to work, only
506          * connected operation is supported, the cnode backvp must
507          * exist, and cachefs optional (eg., disconnectable) flags
508          * are turned off. Assert these conditions to ensure that
509          * the backfilesystem is called for the close operation.
510          */
511         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
512         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);
513 
514         /*
515          * File could have been passed in or inherited from the global zone, so
516          * we don't want to flat out reject the request; we'll just leave things
517          * the way they are and let the backfs (NFS) deal with it.
518          */
519         /* get rid of any local locks */
520         if (CFS_ISFS_LLOCK(fscp)) {
521                 (void) cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
522         }
```

```
524         /* clean up if this is the daemon closing down */
525         if ((fscp->fs_cddaemonid == ttoproc(curthread)->p_pid) &&
526             ((ttoproc(curthread)->p_pid) != 0) &&
527             (vp == fscp->fs_rootvp) &&
528             (count == 1)) {
529                 mutex_enter(&fscp->fs_cdlock);
530                 fscp->fs_cddaemonid = 0;
531                 if (fscp->fs_dlogfile)
532                         fscp->fs_cdconnected = CFS_CD_DISCONNECTED;
533                 else
534                         fscp->fs_cdconnected = CFS_CD_CONNECTED;
535                 cv_broadcast(&fscp->fs_cdwaitcv);
536                 mutex_exit(&fscp->fs_cdlock);
537                 if (fscp->fs_flags & CFS_FS_ROOTFS) {
538                         cachep = fscp->fs_cache;
539                         mutex_enter(&cachep->c_contentslock);
540                         ASSERT(cachep->c_rootdaemonid != 0);
541                         cachep->c_rootdaemonid = 0;
542                         mutex_exit(&cachep->c_contentslock);
543                 }
544                 return (0);
545         }

547         for (;;) {
548                 /* get (or renew) access to the file system */
549                 if (held) {
550                         /* Won't loop with NFSv4 connected behavior */
551                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
552                         cachefs_cd_release(fscp);
553                         held = 0;
554                 }
555                 error = cachefs_cd_access(fscp, connected, 0);
556                 if (error)
557                         goto out;
558                 held = 1;
559                 connected = 0;

561                 /* if not the last close */
562                 if (count > 1) {
563                         if (fscp->fs_cdconnected != CFS_CD_CONNECTED)
564                                 goto out;
565                         mutex_enter(&cp->c_statelock);
566                         if (cp->c_backvp) {
567                                 CFS_DPRINT_BACKFS_NFSV4(fscp,
568                                     ("cachefs_close (nfsv4): cnode %p, "
569                                     "backvp %p\n", cp, cp->c_backvp));
570                                 error = VOP_CLOSE(cp->c_backvp, flag, count,
571                                     offset, cr, ct);
572                                 if (CFS_TIMEOUT(fscp, error)) {
573                                         mutex_exit(&cp->c_statelock);
574                                         cachefs_cd_release(fscp);
575                                         held = 0;
576                                         cachefs_cd_timedout(fscp);
577                                         continue;
578                                 }
579                         }
580                         mutex_exit(&cp->c_statelock);
581                         goto out;
582                 }

584                 /*
585                  * If the file is an unlinked file, then flush the lookup
586                  * cache so that inactive will be called if this is
587                  * the last reference.  It will invalidate all of the
588                  * cached pages, without writing them out.  Writing them
589                  * out is not required because they will be written to a
```



```
590                  * file which will be immediately removed.
591                  */
592                 if (cp->c_unldvp != NULL) {
593                         dnlc_purge_vp(vp);
594                         mutex_enter(&cp->c_statelock);
595                         error = cp->c_error;
596                         cp->c_error = 0;
597                         mutex_exit(&cp->c_statelock);
598                         /* always call VOP_CLOSE() for back fs vnode */
599                 }

601                 /* force dirty data to stable storage */
602                 else if ((vp->v_type == VREG) && (flag & FWRITE) &&
603                     !CFS_ISFS_BACKFS_NFSV4(fscp)) {
604                         /* clean the cachefs pages synchronously */
605                         error = cachefs_putpage_common(vp, (offset_t)0,
606                             0, 0, cr);
607                         if (CFS_TIMEOUT(fscp, error)) {
608                                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
609                                         cachefs_cd_release(fscp);
610                                         held = 0;
611                                         cachefs_cd_timedout(fscp);
612                                         continue;
613                                 } else {
614                                         connected = 1;
615                                         continue;
616                                 }
617                         }

619                         /* if no space left in cache, wait until connected */
620                         if ((error == ENOSPC) &&
621                             (fscp->fs_cdconnected != CFS_CD_CONNECTED)) {
622                                 connected = 1;
623                                 continue;
624                         }

626                         /* clear the cnode error if putpage worked */
627                         if ((error == 0) && cp->c_error) {
628                                 mutex_enter(&cp->c_statelock);
629                                 cp->c_error = 0;
630                                 mutex_exit(&cp->c_statelock);
631                         }

633                         /* if any other important error */
634                         if (cp->c_error) {
635                                 /* get rid of the pages */
636                                 (void) cachefs_putpage_common(vp,
637                                     (offset_t)0, 0, B_INVAL | B_FORCE, cr);
638                                 dnlc_purge_vp(vp);
639                         }
640                 }

642                 mutex_enter(&cp->c_statelock);
643                 if (cp->c_backvp &&
644                     (fscp->fs_cdconnected == CFS_CD_CONNECTED)) {
645                         error = VOP_CLOSE(cp->c_backvp, flag, close_cnt,
646                             offset, cr, ct);
647                         if (CFS_TIMEOUT(fscp, error)) {
648                                 mutex_exit(&cp->c_statelock);
649                                 cachefs_cd_release(fscp);
650                                 held = 0;
651                                 cachefs_cd_timedout(fscp);
652                                 /* don't decrement the vnode counts again */
653                                 close_cnt = 0;
654                                 continue;
655                         }
```

```
656                 }
657                 mutex_exit(&cp->c_statelock);
658                 break;
659         }

661         mutex_enter(&cp->c_statelock);
662         if (!error)
663                 error = cp->c_error;
664         cp->c_error = 0;
665         mutex_exit(&cp->c_statelock);

667 out:
668         if (held)
669                 cachefs_cd_release(fscp);
670 #ifdef CFS_CD_DEBUG
671         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
672 #endif

674 #ifdef CFSDEBUG
675         CFS_DEBUG(CFSDEBUG_VOPS)
676                 printf("cachefs_close: EXIT vp %p\n", (void *)vp);
677 #endif
678         return (error);
679 }

681 /*ARGSUSED*/
682 static int
683 cachefs_read(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
684         caller_context_t *ct)
685 {
686         struct cnode *cp = VTOC(vp);
687         fscache_t *fscp = C_TO_FSCACHE(cp);
688         register u_offset_t off;
689         register int mapoff;
690         register caddr_t base;
691         int n;
692         offset_t diff;
693         uint_t flags = 0;
694         int error = 0;

696 #if 0
697         if (vp->v_flag & VNOCACHE)
698                 flags = SM_INVAL;
699 #endif
700         if (getzoneid() != GLOBAL_ZONEID)
701                 return (EPERM);
702         if (vp->v_type != VREG)
703                 return (EISDIR);

705         ASSERT(RW_READ_HELD(&cp->c_rwlock));

707         if (uiop->uio_resid == 0)
708                 return (0);


711         if (uiop->uio_loffset < (offset_t)0)
712                 return (EINVAL);

714         /*
715          * Call backfilesystem to read if NFSv4, the cachefs code
716          * does the read from the back filesystem asynchronously
717          * which is not supported by pass-through functionality.
718          */
719         if (CFS_ISFS_BACKFS_NFSV4(fscp)) {
720                 error = cachefs_read_backfs_nfsv4(vp, uiop, ioflag, cr, ct);
721                 goto out;
```



```
722         }

724         if (MANDLOCK(vp, cp->c_attr.va_mode)) {
725                 error = chklock(vp, FREAD, (offset_t)uiop->uio_loffset,
726                     uiop->uio_resid, uiop->uio_fmode, ct);
727                 if (error)
728                         return (error);
729         }

731         /*
732          * Sit in a loop and transfer (uiomove) the data in up to
733          * MAXBSIZE chunks. Each chunk is mapped into the kernel's
734          * address space as needed and then released.
735          */
736         do {
737                 /*
738                  *      off     Offset of current MAXBSIZE chunk
739                  *      mapoff  Offset within the current chunk
740                  *      n       Number of bytes to move from this chunk
741                  *      base    kernel address of mapped in chunk
742                  */
743                 off = uiop->uio_loffset & (offset_t)MAXBMASK;
744                 mapoff = uiop->uio_loffset & MAXBOFFSET;
745                 n = MAXBSIZE - mapoff;
746                 if (n > uiop->uio_resid)
747                         n = (uint_t)uiop->uio_resid;

749                 /* perform consistency check */
750                 error = cachefs_cd_access(fscp, 0, 0);
751                 if (error)
752                         break;
753                 mutex_enter(&cp->c_statelock);
754                 error = CFSOP_CHECK_COBJECT(fscp, cp, 0, cr);
755                 diff = cp->c_size - uiop->uio_loffset;
756                 mutex_exit(&cp->c_statelock);
757                 if (CFS_TIMEOUT(fscp, error)) {
758                         cachefs_cd_release(fscp);
759                         cachefs_cd_timedout(fscp);
760                         error = 0;
761                         continue;
762                 }
763                 cachefs_cd_release(fscp);

765                 if (error)
766                         break;

768                 if (diff <= (offset_t)0)
769                         break;
770                 if (diff < (offset_t)n)
771                         n = diff;

773                 base = segmap_getmapflt(segkmap, vp, off, (uint_t)n, 1, S_READ);

775                 error = segmap_fault(kas.a_hat, segkmap, base, n,
776                     F_SOFTLOCK, S_READ);
777                 if (error) {
778                         (void) segmap_release(segkmap, base, 0);
779                         if (FC_CODE(error) == FC_OBJERR)
780                                 error =  FC_ERRNO(error);
781                         else
782                                 error = EIO;
783                         break;
784                 }
785                 error = uiomove(base+mapoff, n, UIO_READ, uiop);
786                 (void) segmap_fault(kas.a_hat, segkmap, base, n,
787                     F_SOFTUNLOCK, S_READ);
```

```
788                 if (error == 0) {
789                         /*
790                          * if we read a whole page(s), or to eof,
791                          *  we won't need this page(s) again soon.
792                          */
793                         if (n + mapoff == MAXBSIZE ||
794                             uiop->uio_loffset == cp->c_size)
795                                 flags |= SM_DONTNEED;
796                 }
797                 (void) segmap_release(segkmap, base, flags);
798         } while (error == 0 && uiop->uio_resid > 0);

800 out:
801 #ifdef CFSDEBUG
802         CFS_DEBUG(CFSDEBUG_VOPS)
803                 printf("cachefs_read: EXIT error %d resid %ld\n", error,
804                     uiop->uio_resid);
805 #endif
806         return (error);
807 }

809 /*
810  * cachefs_read_backfs_nfsv4
811  *
812  * Call NFSv4 back filesystem to handle the read (cachefs
813  * pass-through support for NFSv4).
814  */
815 static int
816 cachefs_read_backfs_nfsv4(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
817                         caller_context_t *ct)
818 {
819         cnode_t *cp = VTOC(vp);
820         fscache_t *fscp = C_TO_FSCACHE(cp);
821         vnode_t *backvp;
822         int error;

824         /*
825          * For NFSv4 pass-through to work, only connected operation
826          * is supported, the cnode backvp must exist, and cachefs
827          * optional (eg., disconnectable) flags are turned off. Assert
828          * these conditions for the read operation.
829          */
830         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
831         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

833         /* Call backfs vnode op after extracting backvp */
834         mutex_enter(&cp->c_statelock);
835         backvp = cp->c_backvp;
836         mutex_exit(&cp->c_statelock);

838         CFS_DPRINT_BACKFS_NFSV4(fscp, ("cachefs_read_backfs_nfsv4: cnode %p, "
839             "backvp %p\n", cp, backvp));

841         (void) VOP_RWLOCK(backvp, V_WRITELOCK_FALSE, ct);
842         error = VOP_READ(backvp, uiop, ioflag, cr, ct);
843         VOP_RWUNLOCK(backvp, V_WRITELOCK_FALSE, ct);

845         /* Increment cache miss counter */
846         fscp->fs_stats.st_misses++;

848         return (error);
849 }

851 /*ARGSUSED*/
852 static int
853 cachefs_write(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
```



```
854         caller_context_t *ct)
855 {
856         struct cnode *cp = VTOC(vp);
857         fscache_t *fscp = C_TO_FSCACHE(cp);
858         int error = 0;
859         u_offset_t off;
860         caddr_t base;
861         uint_t bsize;
862         uint_t flags;
863         int n, on;
864         rlim64_t limit = uiop->uio_llimit;
865         ssize_t resid;
866         offset_t offset;
867         offset_t remainder;

869 #ifdef CFSDEBUG
870         CFS_DEBUG(CFSDEBUG_VOPS)
871                 printf(
872                 "cachefs_write: ENTER vp %p offset %llu count %ld cflags %x\n",
873                     (void *)vp, uiop->uio_loffset, uiop->uio_resid,
874                     cp->c_flags);
875 #endif
876         if (getzoneid() != GLOBAL_ZONEID) {
877                 error = EPERM;
878                 goto out;
879         }
880         if (vp->v_type != VREG) {
881                 error = EISDIR;
882                 goto out;
883         }

885         ASSERT(RW_WRITE_HELD(&cp->c_rwlock));

887         if (uiop->uio_resid == 0) {
888                 goto out;
889         }

891         /* Call backfilesystem to write if NFSv4 */
892         if (CFS_ISFS_BACKFS_NFSV4(fscp)) {
893                 error = cachefs_write_backfs_nfsv4(vp, uiop, ioflag, cr, ct);
894                 goto out2;
895         }

897         if (MANDLOCK(vp, cp->c_attr.va_mode)) {
898                 error = chklock(vp, FWRITE, (offset_t)uiop->uio_loffset,
899                     uiop->uio_resid, uiop->uio_fmode, ct);
900                 if (error)
901                         goto out;
902         }

904         if (ioflag & FAPPEND) {
905                 for (;;) {
906                         /* do consistency check to get correct file size */
907                         error = cachefs_cd_access(fscp, 0, 1);
908                         if (error)
909                                 goto out;
910                         mutex_enter(&cp->c_statelock);
911                         error = CFSOP_CHECK_COBJECT(fscp, cp, 0, cr);
912                         uiop->uio_loffset = cp->c_size;
913                         mutex_exit(&cp->c_statelock);
914                         if (CFS_TIMEOUT(fscp, error)) {
915                                 cachefs_cd_release(fscp);
916                                 cachefs_cd_timedout(fscp);
917                                 continue;
918                         }
919                         cachefs_cd_release(fscp);
```

```
920                         if (error)
921                                 goto out;
922                         break;
923                 }
924         }

926         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
927                 limit = MAXOFFSET_T;

929         if (uiop->uio_loffset >= limit) {
930                 proc_t *p = ttoproc(curthread);

932                 mutex_enter(&p->p_lock);
933                 (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE], p->p_rctls,
934                     p, RCA_UNSAFE_SIGINFO);
935                 mutex_exit(&p->p_lock);
936                 error = EFBIG;
937                 goto out;
938         }
939         if (uiop->uio_loffset > fscp->fs_offmax) {
940                 error = EFBIG;
941                 goto out;
942         }

944         if (limit > fscp->fs_offmax)
945                 limit = fscp->fs_offmax;

947         if (uiop->uio_loffset < (offset_t)0) {
948                 error = EINVAL;
949                 goto out;
950         }

952         offset = uiop->uio_loffset + uiop->uio_resid;
953         /*
954          * Check to make sure that the process will not exceed
955          * its limit on file size.  It is okay to write up to
956          * the limit, but not beyond.  Thus, the write which
957          * reaches the limit will be short and the next write
958          * will return an error.
959          */
960         remainder = 0;
961         if (offset > limit) {
962                 remainder = (int)(offset - (u_offset_t)limit);
963                 uiop->uio_resid = limit - uiop->uio_loffset;
964                 if (uiop->uio_resid <= 0) {
965                         proc_t *p = ttoproc(curthread);

967                         uiop->uio_resid += remainder;
968                         mutex_enter(&p->p_lock);
969                         (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE],
970                             p->p_rctls, p, RCA_UNSAFE_SIGINFO);
971                         mutex_exit(&p->p_lock);
972                         error = EFBIG;
973                         goto out;
974                 }
975         }

977         resid = uiop->uio_resid;
978         offset = uiop->uio_loffset;
979         bsize = vp->v_vfsp->vfs_bsize;

981         /* loop around and do the write in MAXBSIZE chunks */
982         do {
983                 /* mapping offset */
984                 off = uiop->uio_loffset & (offset_t)MAXBMASK;
985                 on = uiop->uio_loffset & MAXBOFFSET; /* Rel. offset */
```



```
986                 n = MAXBSIZE - on;
987                 if (n > uiop->uio_resid)
988                         n = (int)uiop->uio_resid;

990                 /*
991                  * Touch the page and fault it in if it is not in
992                  * core before segmap_getmapflt can lock it. This
993                  * is to avoid the deadlock if the buffer is mapped
994                  * to the same file through mmap which we want to
995                  * write to.
996                  */
997                 uio_prefaultpages((long)n, uiop);

999                 base = segmap_getmap(segkmap, vp, off);
1000                error = cachefs_writepage(vp, (base + on), n, uiop);
1001                if (error == 0) {
1002                        flags = 0;
1003                        /*
1004                         * Have written a whole block.Start an
1005                         * asynchronous write and mark the buffer to
1006                         * indicate that it won't be needed again
1007                         * soon.
1008                         */
1009                        if (n + on == bsize) {
1010                                flags = SM_WRITE |SM_ASYNC |SM_DONTNEED;
1011                        }
1012 #if 0
1013                        /* XXX need to understand this */
1014                        if ((ioflag & (FSYNC|FDSYNC)) ||
1015                            (cp->c_backvp && vn_has_flocks(cp->c_backvp))) {
1016                                flags &= ~SM_ASYNC;
1017                                flags |= SM_WRITE;
1018                        }
1019 #else
1020                        if (ioflag & (FSYNC|FDSYNC)) {
1021                                flags &= ~SM_ASYNC;
1022                                flags |= SM_WRITE;
1023                        }
1024 #endif
1025                        error = segmap_release(segkmap, base, flags);
1026                } else {
1027                        (void) segmap_release(segkmap, base, 0);
1028                }
1029        } while (error == 0 && uiop->uio_resid > 0);

1031 out:
1032        if (error == EINTR && (ioflag & (FSYNC|FDSYNC))) {
1033                uiop->uio_resid = resid;
1034                uiop->uio_loffset = offset;
1035        } else
1036                uiop->uio_resid += remainder;

1038 out2:
1039 #ifdef CFSDEBUG
1040        CFS_DEBUG(CFSDEBUG_VOPS)
1041                printf("cachefs_write: EXIT error %d\n", error);
1042 #endif
1043        return (error);
1044 }

1046 /*
1047  * cachefs_write_backfs_nfsv4
1048  *
1049  * Call NFSv4 back filesystem to handle the write (cachefs
1050  * pass-through support for NFSv4).
1051  */
```

```
1052 static int
1053 cachefs_write_backfs_nfsv4(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
1054                         caller_context_t *ct)
1055 {
1056         cnode_t *cp = VTOC(vp);
1057         fscache_t *fscp = C_TO_FSCACHE(cp);
1058         vnode_t *backvp;
1059         int error;
1060
1061         /*
1062          * For NFSv4 pass-through to work, only connected operation
1063          * is supported, the cnode backvp must exist, and cachefs
1064          * optional (eg., disconnectable) flags are turned off. Assert
1065          * these conditions for the read operation.
1066          */
1067         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
1068         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);
1069
1070         /* Call backfs vnode op after extracting the backvp */
1071         mutex_enter(&cp->c_statelock);
1072         backvp = cp->c_backvp;
1073         mutex_exit(&cp->c_statelock);
1074
1075         CFS_DPRINT_BACKFS_NFSV4(fscp, ("cachefs_write_backfs_nfsv4: cnode %p, "
1076             "backvp %p\n", cp, backvp));
1077         (void) VOP_RWLOCK(backvp, V_WRITELOCK_TRUE, ct);
1078         error = VOP_WRITE(backvp, uiop, ioflag, cr, ct);
1079         VOP_RWUNLOCK(backvp, V_WRITELOCK_TRUE, ct);
1080
1081         return (error);
1082 }
1083
1084 /*
1085  * see if we've charged ourselves for frontfile data at
1086  * the given offset.  If not, allocate a block for it now.
1087  */
1088 static int
1089 cachefs_charge_page(struct cnode *cp, u_offset_t offset)
1090 {
1091         u_offset_t blockoff;
1092         int error;
1093         int inc;
1094
1095         ASSERT(MUTEX_HELD(&cp->c_statelock));
1096         /*LINTED*/
1097         ASSERT(PAGESIZE <= MAXBSIZE);
1098
1099         error = 0;
1100         blockoff = offset & (offset_t)MAXBMASK;
1101
1102         /* get the front file if necessary so allocblocks works */
1103         if ((cp->c_frontvp == NULL) &&
1104             ((cp->c_flags & CN_NOCACHE) == 0)) {
1105                 (void) cachefs_getfrontfile(cp);
1106         }
1107         if (cp->c_flags & CN_NOCACHE)
1108                 return (1);
1109
1110         if (cachefs_check_allocmap(cp, blockoff))
1111                 return (0);
1112
1113         for (inc = PAGESIZE; inc < MAXBSIZE; inc += PAGESIZE)
1114                 if (cachefs_check_allocmap(cp, blockoff+inc))
1115                         return (0);
1116
1117         error = cachefs_allocblocks(C_TO_FSCACHE(cp)->fs_cache, 1,
```



```
1118             cp->c_metadata.md_rltype);
1119         if (error == 0) {
1120                 cp->c_metadata.md_frontblks++;
1121                 cp->c_flags |= CN_UPDATED;
1122         }
1123         return (error);
1124 }
1125
1126 /*
1127  * Called only by cachefs_write to write 1 page or less of data.
1128  *      base   - base address kernel addr space
1129  *      tcount - Total bytes to move - < MAXBSIZE
1130  */
1131 static int
1132 cachefs_writepage(vnode_t *vp, caddr_t base, int tcount, uio_t *uiop)
1133 {
1134         struct cnode *cp =  VTOC(vp);
1135         fscache_t *fscp = C_TO_FSCACHE(cp);
1136         register int n;
1137         register u_offset_t offset;
1138         int error = 0, terror;
1139         extern struct as kas;
1140         u_offset_t lastpage_off;
1141         int pagecreate = 0;
1142         int newpage;
1143
1144 #ifdef CFSDEBUG
1145         CFS_DEBUG(CFSDEBUG_VOPS)
1146                 printf(
1147                     "cachefs_writepage: ENTER vp %p offset %llu len %ld\\\n",
1148                     (void *)vp, uiop->uio_loffset, uiop->uio_resid);
1149 #endif
1150
1151         /*
1152          * Move bytes in PAGESIZE chunks. We must avoid spanning pages in
1153          * uiomove() because page faults may cause the cache to be invalidated
1154          * out from under us.
1155          */
1156         do {
1157                 offset = uiop->uio_loffset;
1158                 lastpage_off = (cp->c_size - 1) & (offset_t)PAGEMASK;
1159
1160                 /*
1161                  * If not connected then need to make sure we have space
1162                  * to perform the write.  We could make this check
1163                  * a little tighter by only doing it if we are growing the file.
1164                  */
1165                 if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
1166                         error = cachefs_allocblocks(fscp->fs_cache, 1,
1167                             cp->c_metadata.md_rltype);
1168                         if (error)
1169                                 break;
1170                         cachefs_freeblocks(fscp->fs_cache, 1,
1171                             cp->c_metadata.md_rltype);
1172                 }
1173
1174                 /*
1175                  * n is the number of bytes required to satisfy the request
1176                  * or the number of bytes to fill out the page.
1177                  */
1178                 n = (int)(PAGESIZE - ((uintptr_t)base & PAGEOFFSET));
1179                 if (n > tcount)
1180                         n = tcount;
1181
1182                 /*
1183                  * The number of bytes of data in the last page can not
```

```
1184                 * be accurately be determined while page is being
1185                 * uiomove'd to and the size of the file being updated.
1186                 * Thus, inform threads which need to know accurately
1187                 * how much data is in the last page of the file.  They
1188                 * will not do the i/o immediately, but will arrange for
1189                 * the i/o to happen later when this modify operation
1190                 * will have finished.
1191                 *
1192                 * in similar NFS code, this is done right before the
1193                 * uiomove(), which is best.  but here in cachefs, we
1194                 * have two uiomove()s, so we must do it here.
1195                 */
1196                ASSERT(!(cp->c_flags & CN_CMODINPROG));
1197                mutex_enter(&cp->c_statelock);
1198                cp->c_flags |= CN_CMODINPROG;
1199                cp->c_modaddr = (offset & (offset_t)MAXBMASK);
1200                mutex_exit(&cp->c_statelock);

1202                /*
1203                 * Check to see if we can skip reading in the page
1204                 * and just allocate the memory.  We can do this
1205                 * if we are going to rewrite the entire mapping
1206                 * or if we are going to write to or beyond the current
1207                 * end of file from the beginning of the mapping.
1208                 */
1209                if ((offset > (lastpage_off + PAGEOFFSET)) ||
1210                    ((cp->c_size == 0) && (offset < PAGESIZE)) ||
1211                    ((uintptr_t)base & PAGEOFFSET) == 0 && (n == PAGESIZE ||
1212                    ((offset + n) >= cp->c_size))) {
1213                        pagecreate = 1;

1215                        /*
1216                         * segmap_pagecreate() returns 1 if it calls
1217                         * page_create_va() to allocate any pages.
1218                         */
1219                        newpage = segmap_pagecreate(segkmap,
1220                            (caddr_t)((uintptr_t)base & (uintptr_t)PAGEMASK),
1221                            PAGESIZE, 0);
1222                        /* do not zero page if we are overwriting all of it */
1223                        if (!((((uintptr_t)base & PAGEOFFSET) == 0) &&
1224                            (n == PAGESIZE))) {
1225                                (void) kzero((void *)
1226                                    ((uintptr_t)base & (uintptr_t)PAGEMASK),
1227                                    PAGESIZE);
1228                        }
1229                        error = uiomove(base, n, UIO_WRITE, uiop);

1231                        /*
1232                         * Unlock the page allocated by page_create_va()
1233                         * in segmap_pagecreate()
1234                         */
1235                        if (newpage)
1236                                segmap_pageunlock(segkmap,
1237                                    (caddr_t)((uintptr_t)base &
1238                                    (uintptr_t)PAGEMASK),
1239                                    PAGESIZE, S_WRITE);
1240                } else {
1241                        /*
1242                         * KLUDGE ! Use segmap_fault instead of faulting and
1243                         * using as_fault() to avoid a recursive readers lock
1244                         * on kas.
1245                         */
1246                        error = segmap_fault(kas.a_hat, segkmap, (caddr_t)
1247                            ((uintptr_t)base & (uintptr_t)PAGEMASK),
1248                            PAGESIZE, F_SOFTLOCK, S_WRITE);
1249                        if (error) {
```



```
1250                                if (FC_CODE(error) == FC_OBJERR)
1251                                        error =  FC_ERRNO(error);
1252                                else
1253                                        error = EIO;
1254                                break;
1255                        }
1256                        error = uiomove(base, n, UIO_WRITE, uiop);
1257                        (void) segmap_fault(kas.a_hat, segkmap, (caddr_t)
1258                            ((uintptr_t)base & (uintptr_t)PAGEMASK),
1259                            PAGESIZE, F_SOFTUNLOCK, S_WRITE);
1260                }
1261                n = (int)(uiop->uio_loffset - offset); /* n = # bytes written */
1262                base += n;
1263                tcount -= n;

1265                /* get access to the file system */
1266                if ((terror = cachefs_cd_access(fscp, 0, 1)) != 0) {
1267                        error = terror;
1268                        break;
1269                }

1271                /*
1272                 * cp->c_attr.va_size is the maximum number of
1273                 * bytes known to be in the file.
1274                 * Make sure it is at least as high as the
1275                 * last byte we just wrote into the buffer.
1276                 */
1277                mutex_enter(&cp->c_statelock);
1278                if (cp->c_size < uiop->uio_loffset) {
1279                        cp->c_size = uiop->uio_loffset;
1280                }
1281                if (cp->c_size != cp->c_attr.va_size) {
1282                        cp->c_attr.va_size = cp->c_size;
1283                        cp->c_flags |= CN_UPDATED;
1284                }
1285                /* c_size is now correct, so we can clear modinprog */
1286                cp->c_flags &= ~CN_CMODINPROG;
1287                if (error == 0) {
1288                        cp->c_flags |= CDIRTY;
1289                        if (pagecreate && (cp->c_flags & CN_NOCACHE) == 0) {
1290                                /*
1291                                 * if we're not in NOCACHE mode
1292                                 * (i.e., single-writer), we update the
1293                                 * allocmap here rather than waiting until
1294                                 * cachefspush is called.  This prevents
1295                                 * getpage from clustering up pages from
1296                                 * the backfile and stomping over the changes
1297                                 * we make here.
1298                                 */
1299                                if (cachefs_charge_page(cp, offset) == 0) {
1300                                        cachefs_update_allocmap(cp,
1301                                            offset & (offset_t)PAGEMASK,
1302                                            (size_t)PAGESIZE);
1303                                }

1305                                /* else we ran out of space */
1306                                else {
1307                                        /* nocache file if connected */
1308                                        if (fscp->fs_cdconnected ==
1309                                            CFS_CD_CONNECTED)
1310                                                cachefs_nocache(cp);
1311                                        /*
1312                                         * If disconnected then cannot
1313                                         * nocache the file.  Let it have
1314                                         * the space.
1315                                         */
```

```
1316                                    else {
1317                                            cp->c_metadata.md_frontblks++;
1318                                            cp->c_flags |= CN_UPDATED;
1319                                            cachefs_update_allocmap(cp,
1320                                                offset & (offset_t)PAGEMASK,
1321                                                (size_t)PAGESIZE);
1322                                    }
1323                                }
1324                        }
1325                }
1326                mutex_exit(&cp->c_statelock);
1327                cachefs_cd_release(fscp);
1328        } while (tcount > 0 && error == 0);

1330        if (cp->c_flags & CN_CMODINPROG) {
1331                /* XXX assert error != 0?  FC_ERRNO() makes this more risky. */
1332                mutex_enter(&cp->c_statelock);
1333                cp->c_flags &= ~CN_CMODINPROG;
1334                mutex_exit(&cp->c_statelock);
1335        }

1337 #ifdef CFS_CD_DEBUG
1338        ASSERT((curthread->t_flag & T_CD_HELD) == 0);
1339 #endif

1341 #ifdef CFSDEBUG
1342        CFS_DEBUG(CFSDEBUG_VOPS)
1343                printf("cachefs_writepage: EXIT error %d\n", error);
1344 #endif

1346        return (error);
1347 }

1349 /*
1350  * Pushes out pages to the back and/or front file system.
1351  */
1352 static int
1353 cachefs_push(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *lenp,
1354     int flags, cred_t *cr)
1355 {
1356        struct cnode *cp = VTOC(vp);
1357        struct buf *bp;
1358        int error;
1359        fscache_t *fscp = C_TO_FSCACHE(cp);
1360        u_offset_t iooff;
1361        size_t iolen;
1362        u_offset_t lbn;
1363        u_offset_t lbn_off;
1364        uint_t bsize;

1366        ASSERT((flags & B_ASYNC) == 0);
1367        ASSERT(!vn_is_readonly(vp));
1368        ASSERT(pp != NULL);
1369        ASSERT(cr != NULL);

1371        bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
1372        lbn = pp->p_offset / bsize;
1373        lbn_off = lbn * bsize;

1375        /*
1376         * Find a kluster that fits in one block, or in
1377         * one page if pages are bigger than blocks.  If
1378         * there is less file space allocated than a whole
1379         * page, we'll shorten the i/o request below.
1380         */
```



```
1382        pp = pvn_write_kluster(vp, pp, &iooff, &iolen, lbn_off,
1383            roundup(bsize, PAGESIZE), flags);

1385        /*
1386         * The CN_CMODINPROG flag makes sure that we use a correct
1387         * value of c_size, below.  CN_CMODINPROG is set in
1388         * cachefs_writepage().  When CN_CMODINPROG is set it
1389         * indicates that a uiomove() is in progress and the c_size
1390         * has not been made consistent with the new size of the
1391         * file. When the uiomove() completes the c_size is updated
1392         * and the CN_CMODINPROG flag is cleared.
1393         *
1394         * The CN_CMODINPROG flag makes sure that cachefs_push_front
1395         * and cachefs_push_connected see a consistent value of
1396         * c_size.  Without this handshaking, it is possible that
1397         * these routines will pick up the old value of c_size before
1398         * the uiomove() in cachefs_writepage() completes.  This will
1399         * result in the vn_rdwr() being too small, and data loss.
1400         *
1401         * More precisely, there is a window between the time the
1402         * uiomove() completes and the time the c_size is updated. If
1403         * a VOP_PUTPAGE() operation intervenes in this window, the
1404         * page will be picked up, because it is dirty; it will be
1405         * unlocked, unless it was pagecreate'd. When the page is
1406         * picked up as dirty, the dirty bit is reset
1407         * (pvn_getdirty()). In cachefs_push_connected(), c_size is
1408         * checked.  This will still be the old size.  Therefore, the
1409         * page will not be written out to the correct length, and the
1410         * page will be clean, so the data may disappear.
1411         */
1412        if (cp->c_flags & CN_CMODINPROG) {
1413                mutex_enter(&cp->c_statelock);
1414                if ((cp->c_flags & CN_CMODINPROG) &&
1415                    cp->c_modaddr + MAXBSIZE > iooff &&
1416                    cp->c_modaddr < iooff + iolen) {
1417                        page_t *plist;

1419                        /*
1420                         * A write is in progress for this region of
1421                         * the file.  If we did not detect
1422                         * CN_CMODINPROG here then this path through
1423                         * cachefs_push_connected() would eventually
1424                         * do the vn_rdwr() and may not write out all
1425                         * of the data in the pages.  We end up losing
1426                         * data. So we decide to set the modified bit
1427                         * on each page in the page list and mark the
1428                         * cnode with CDIRTY.  This push will be
1429                         * restarted at some later time.
1430                         */

1432                        plist = pp;
1433                        while (plist != NULL) {
1434                                pp = plist;
1435                                page_sub(&plist, pp);
1436                                hat_setmod(pp);
1437                                page_io_unlock(pp);
1438                                page_unlock(pp);
1439                        }
1440                        cp->c_flags |= CDIRTY;
1441                        mutex_exit(&cp->c_statelock);
1442                        if (offp)
1443                                *offp = iooff;
1444                        if (lenp)
1445                                *lenp = iolen;
1446                        return (0);
1447                }
```

```
1448                    mutex_exit(&cp->c_statelock);
1449            }

1451            /*
1452             * Set the pages up for pageout.
1453             */
1454            bp = pageio_setup(pp, iolen, CTOV(cp), B_WRITE | flags);
1455            if (bp == NULL) {

1457                    /*
1458                     * currently, there is no way for pageio_setup() to
1459                     * return NULL, since it uses its own scheme for
1460                     * kmem_alloc()ing that shouldn't return NULL, and
1461                     * since pageio_setup() itself dereferences the thing
1462                     * it's about to return.  still, we need to be ready
1463                     * in case this ever does start happening.
1464                     */

1466                    error = ENOMEM;
1467                    goto writedone;
1468            }
1469            /*
1470             * pageio_setup should have set b_addr to 0.  This
1471             * is correct since we want to do I/O on a page
1472             * boundary.  bp_mapin will use this addr to calculate
1473             * an offset, and then set b_addr to the kernel virtual
1474             * address it allocated for us.
1475             */
1476            bp->b_edev = 0;
1477            bp->b_dev = 0;
1478            bp->b_lblkno = (diskaddr_t)lbtodb(iooff);
1479            bp_mapin(bp);

1481            iolen  = cp->c_size - ldbtob(bp->b_blkno);
1482            if (iolen > bp->b_bcount)
1483                    iolen  = bp->b_bcount;

1485            /* if connected */
1486            if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
1487                    /* write to the back file first */
1488                    error = cachefs_push_connected(vp, bp, iolen, iooff, cr);

1490                    /* write to the front file if allowed */
1491                    if ((error == 0) && CFS_ISFS_NONSHARED(fscp) &&
1492                        ((cp->c_flags & CN_NOCACHE) == 0)) {
1493                            /* try to write to the front file */
1494                            (void) cachefs_push_front(vp, bp, iolen, iooff, cr);
1495                    }
1496            }

1498            /* else if disconnected */
1499            else {
1500                    /* try to write to the front file */
1501                    error = cachefs_push_front(vp, bp, iolen, iooff, cr);
1502            }

1504            bp_mapout(bp);
1505            pageio_done(bp);

1507    writedone:

1509            pvn_write_done(pp, ((error) ? B_ERROR : 0) | B_WRITE | flags);
1510            if (offp)
1511                    *offp = iooff;
1512            if (lenp)
1513                    *lenp = iolen;
```



```
1515            /* XXX ask bob mastors how to fix this someday */
1516            mutex_enter(&cp->c_statelock);
1517            if (error) {
1518                    if (error == ENOSPC) {
1519                            if ((fscp->fs_cdconnected == CFS_CD_CONNECTED) ||
1520                                CFS_ISFS_SOFT(fscp)) {
1521                                    CFSOP_INVALIDATE_COBJECT(fscp, cp, cr);
1522                                    cp->c_error = error;
1523                            }
1524                    } else if ((CFS_TIMEOUT(fscp, error) == 0) &&
1525                        (error != EINTR)) {
1526                            CFSOP_INVALIDATE_COBJECT(fscp, cp, cr);
1527                            cp->c_error = error;
1528                    }
1529            } else if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
1530                    CFSOP_MODIFY_COBJECT(fscp, cp, cr);
1531            }
1532            mutex_exit(&cp->c_statelock);

1534            return (error);
1535    }

1537    /*
1538     * Pushes out pages to the back file system.
1539     */
1540    static int
1541    cachefs_push_connected(vnode_t *vp, struct buf *bp, size_t iolen,
1542        u_offset_t iooff, cred_t *cr)
1543    {
1544            struct cnode *cp = VTOC(vp);
1545            int error = 0;
1546            int mode = 0;
1547            fscache_t *fscp = C_TO_FSCACHE(cp);
1548            ssize_t resid;
1549            vnode_t *backvp;

1551            /* get the back file if necessary */
1552            mutex_enter(&cp->c_statelock);
1553            if (cp->c_backvp == NULL) {
1554                    error = cachefs_getbackvp(fscp, cp);
1555                    if (error) {
1556                            mutex_exit(&cp->c_statelock);
1557                            goto out;
1558                    }
1559            }
1560            backvp = cp->c_backvp;
1561            VN_HOLD(backvp);
1562            mutex_exit(&cp->c_statelock);

1564            if (CFS_ISFS_NONSHARED(fscp) && CFS_ISFS_SNR(fscp))
1565                    mode = FSYNC;

1567            /* write to the back file */
1568            error = bp->b_error = vn_rdwr(UIO_WRITE, backvp, bp->b_un.b_addr,
1569                iolen, iooff, UIO_SYSSPACE, mode,
1570                RLIM64_INFINITY, cr, &resid);
1571            if (error) {
1572    #ifdef CFSDEBUG
1573                    CFS_DEBUG(CFSDEBUG_VOPS | CFSDEBUG_BACK)
1574                            printf("cachefspush: error %d cr %p\n",
1575                                error, (void *)cr);
1576    #endif
1577                    bp->b_flags |= B_ERROR;
1578            }
1579            VN_RELE(backvp);
```

```
1580 out:
1581         return (error);
1582 }

1584 /*
1585  * Pushes out pages to the front file system.
1586  * Called for both connected and disconnected states.
1587  */
1588 static int
1589 cachefs_push_front(vnode_t *vp, struct buf *bp, size_t iolen,
1590     u_offset_t iooff, cred_t *cr)
1591 {
1592         struct cnode *cp = VTOC(vp);
1593         fscache_t *fscp = C_TO_FSCACHE(cp);
1594         int error = 0;
1595         ssize_t resid;
1596         u_offset_t popoff;
1597         off_t commit = 0;
1598         uint_t seq;
1599         enum cachefs_rl_type type;
1600         vnode_t *frontvp = NULL;

1602         mutex_enter(&cp->c_statelock);

1604         if (!CFS_ISFS_NONSHARED(fscp)) {
1605                 error = ETIMEDOUT;
1606                 goto out;
1607         }

1609         /* get the front file if necessary */
1610         if ((cp->c_frontvp == NULL) &&
1611             ((cp->c_flags & CN_NOCACHE) == 0)) {
1612                 (void) cachefs_getfrontfile(cp);
1613         }
1614         if (cp->c_flags & CN_NOCACHE) {
1615                 error = ETIMEDOUT;
1616                 goto out;
1617         }

1619         /* if disconnected, needs to be populated and have good attributes */
1620         if ((fscp->fs_cdconnected != CFS_CD_CONNECTED) &&
1621             (((cp->c_metadata.md_flags & MD_POPULATED) == 0) ||
1622             (cp->c_metadata.md_flags & MD_NEEDATTRS))) {
1623                 error = ETIMEDOUT;
1624                 goto out;
1625         }

1627         for (popoff = iooff; popoff < (iooff + iolen); popoff += MAXBSIZE) {
1628                 if (cachefs_charge_page(cp, popoff)) {
1629                         if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
1630                                 cachefs_nocache(cp);
1631                                 goto out;
1632                         } else {
1633                                 error = ENOSPC;
1634                                 goto out;
1635                         }
1636                 }
1637         }

1639         if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
1640                 /* log the first putpage to a file */
1641                 if ((cp->c_metadata.md_flags & MD_PUTPAGE) == 0) {
1642                         /* uses open's creds if we have them */
1643                         if (cp->c_cred)
1644                                 cr = cp->c_cred;
```



```
1646                         if ((cp->c_metadata.md_flags & MD_MAPPING) == 0) {
1647                                 error = cachefs_dlog_cidmap(fscp);
1648                                 if (error) {
1649                                         error = ENOSPC;
1650                                         goto out;
1651                                 }
1652                                 cp->c_metadata.md_flags |= MD_MAPPING;
1653                         }

1655                         commit = cachefs_dlog_modify(fscp, cp, cr, &seq);
1656                         if (commit == 0) {
1657                                 /* out of space */
1658                                 error = ENOSPC;
1659                                 goto out;
1660                         }

1662                         cp->c_metadata.md_seq = seq;
1663                         type = cp->c_metadata.md_rltype;
1664                         cachefs_modified(cp);
1665                         cp->c_metadata.md_flags |= MD_PUTPAGE;
1666                         cp->c_metadata.md_flags &= ~MD_PUSHDONE;
1667                         cp->c_flags |= CN_UPDATED;
1668                 }

1670                 /* subsequent putpages just get a new sequence number */
1671                 else {
1672                         /* but only if it matters */
1673                         if (cp->c_metadata.md_seq != fscp->fs_dlogseq) {
1674                                 seq = cachefs_dlog_seqnext(fscp);
1675                                 if (seq == 0) {
1676                                         error = ENOSPC;
1677                                         goto out;
1678                                 }
1679                                 cp->c_metadata.md_seq = seq;
1680                                 cp->c_flags |= CN_UPDATED;
1681                                 /* XXX maybe should do write_metadata here */
1682                         }
1683                 }
1684         }

1686         frontvp = cp->c_frontvp;
1687         VN_HOLD(frontvp);
1688         mutex_exit(&cp->c_statelock);
1689         error = bp->b_error = vn_rdwr(UIO_WRITE, frontvp,
1690             bp->b_un.b_addr, iolen, iooff, UIO_SYSSPACE, 0,
1691             RLIM64_INFINITY, kcred, &resid);
1692         mutex_enter(&cp->c_statelock);
1693         VN_RELE(frontvp);
1694         frontvp = NULL;
1695         if (error) {
1696                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
1697                         cachefs_nocache(cp);
1698                         error = 0;
1699                         goto out;
1700                 } else {
1701                         goto out;
1702                 }
1703         }

1705         (void) cachefs_update_allocmap(cp, iooff, iolen);
1706         cp->c_flags |= (CN_UPDATED | CN_NEED_FRONT_SYNC |
1707             CN_POPULATION_PENDING);
1708         if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
1709                 gethrestime(&cp->c_metadata.md_localmtime);
1710                 cp->c_metadata.md_flags |= MD_LOCALMTIME;
1711         }
```

```
1713 out:
1714         if (commit) {
1715                 /* commit the log record */
1716                 ASSERT(fscp->fs_cdconnected == CFS_CD_DISCONNECTED);
1717                 if (cachefs_dlog_commit(fscp, commit, error)) {
1718                         /*EMPTY*/
1719                         /* XXX fix on panic */
1720                 }
1721         }

1723         if (error && commit) {
1724                 cp->c_metadata.md_flags &= ~MD_PUTPAGE;
1725                 cachefs_rlent_moveto(fscp->fs_cache, type,
1726                     cp->c_metadata.md_rlno, cp->c_metadata.md_frontblks);
1727                 cp->c_metadata.md_rltype = type;
1728                 cp->c_flags |= CN_UPDATED;
1729         }
1730         mutex_exit(&cp->c_statelock);
1731         return (error);
1732 }

1734 /*ARGSUSED*/
1735 static int
1736 cachefs_dump(struct vnode *vp, caddr_t foo1, offset_t foo2, offset_t foo3,
1737     caller_context_t *ct)
1738 {
1739         return (ENOSYS); /* should we panic if we get here? */
1740 }

1742 /*ARGSUSED*/
1743 static int
1744 cachefs_ioctl(struct vnode *vp, int cmd, intptr_t arg, int flag, cred_t *cred,
1745         int *rvalp, caller_context_t *ct)
1746 {
1747         int error;
1748         struct cnode *cp = VTOC(vp);
1749         struct fscache *fscp = C_TO_FSCACHE(cp);
1750         struct cachefscache *cachep;
1751         extern kmutex_t cachefs_cachelock;
1752         extern cachefscache_t *cachefs_cachelist;
1753         cachefsio_pack_t *packp;
1754         STRUCT_DECL(cachefsio_dcmd, dcmd);
1755         int     inlen, outlen;  /* LP64: generic int for struct in/out len */
1756         void *dinp, *doutp;
1757         int (*dcmd_routine)(vnode_t *, void *, void *);

1759         if (getzoneid() != GLOBAL_ZONEID)
1760                 return (EPERM);

1762         /*
1763          * Cachefs only provides pass-through support for NFSv4,
1764          * and all vnode operations are passed through to the
1765          * back file system. For NFSv4 pass-through to work, only
1766          * connected operation is supported, the cnode backvp must
1767          * exist, and cachefs optional (eg., disconnectable) flags
1768          * are turned off. Assert these conditions which ensure
1769          * that only a subset of the ioctls are "truly supported"
1770          * for NFSv4 (these are CFSDCMD_DAEMONID and CFSDCMD_GETSTATS.
1771          * The packing operations are meaningless since there is
1772          * no caching for NFSv4, and the called functions silently
1773          * return if the backfilesystem is NFSv4. The daemon
1774          * commands except for those above are essentially used
1775          * for disconnectable operation support (including log
1776          * rolling), so in each called function, we assert that
1777          * NFSv4 is not in use. The _FIO* calls (except _FIOCOD)
```



```
1778          * are from "cfsfstype" which is not a documented
1779          * command. However, the command is visible in
1780          * /usr/lib/fs/cachefs so the commands are simply let
1781          * through (don't seem to impact pass-through functionality).
1782          */
1783         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
1784         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

1786         switch (cmd) {
1787         case CACHEFSIO_PACK:
1788                 packp = cachefs_kmem_alloc(sizeof (cachefsio_pack_t), KM_SLEEP);
1789                 error = xcopyin((void *)arg, packp, sizeof (cachefsio_pack_t));
1790                 if (!error)
1791                         error = cachefs_pack(vp, packp->p_name, cred);
1792                 cachefs_kmem_free(packp, sizeof (cachefsio_pack_t));
1793                 break;

1795         case CACHEFSIO_UNPACK:
1796                 packp = cachefs_kmem_alloc(sizeof (cachefsio_pack_t), KM_SLEEP);
1797                 error = xcopyin((void *)arg, packp, sizeof (cachefsio_pack_t));
1798                 if (!error)
1799                         error = cachefs_unpack(vp, packp->p_name, cred);
1800                 cachefs_kmem_free(packp, sizeof (cachefsio_pack_t));
1801                 break;

1803         case CACHEFSIO_PACKINFO:
1804                 packp = cachefs_kmem_alloc(sizeof (cachefsio_pack_t), KM_SLEEP);
1805                 error = xcopyin((void *)arg, packp, sizeof (cachefsio_pack_t));
1806                 if (!error)
1807                         error = cachefs_packinfo(vp, packp->p_name,
1808                             &packp->p_status, cred);
1809                 if (!error)
1810                         error = xcopyout(packp, (void *)arg,
1811                             sizeof (cachefsio_pack_t));
1812                 cachefs_kmem_free(packp, sizeof (cachefsio_pack_t));
1813                 break;

1815         case CACHEFSIO_UNPACKALL:
1816                 error = cachefs_unpackall(vp);
1817                 break;

1819         case CACHEFSIO_DCMD:
1820                 /*
1821                  * This is a private interface between the cachefsd and
1822                  * this file system.
1823                  */

1825                 /* must be root to use these commands */
1826                 if (secpolicy_fs_config(cred, vp->v_vfsp) != 0)
1827                         return (EPERM);

1829                 /* get the command packet */
1830                 STRUCT_INIT(dcmd, flag & DATAMODEL_MASK);
1831                 error = xcopyin((void *)arg, STRUCT_BUF(dcmd),
1832                     SIZEOF_STRUCT(cachefsio_dcmd, DATAMODEL_NATIVE));
1833                 if (error)
1834                         return (error);

1836                 /* copy in the data for the operation */
1837                 dinp = NULL;
1838                 if ((inlen = STRUCT_FGET(dcmd, d_slen)) > 0) {
1839                         dinp = cachefs_kmem_alloc(inlen, KM_SLEEP);
1840                         error = xcopyin(STRUCT_FGETP(dcmd, d_sdata), dinp,
1841                             inlen);
1842                         if (error)
1843                                 return (error);
```

```
1844                }

1846                /* allocate space for the result */
1847                doutp = NULL;
1848                if ((outlen = STRUCT_FGET(dcmd, d_rlen)) > 0)
1849                        doutp = cachefs_kmem_alloc(outlen, KM_SLEEP);

1851                /*
1852                 * Assert NFSv4 only allows the daemonid and getstats
1853                 * daemon requests
1854                 */
1855                ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0 ||
1856                    STRUCT_FGET(dcmd, d_cmd) == CFSDCMD_DAEMONID ||
1857                    STRUCT_FGET(dcmd, d_cmd) == CFSDCMD_GETSTATS);

1859                /* get the routine to execute */
1860                dcmd_routine = NULL;
1861                switch (STRUCT_FGET(dcmd, d_cmd)) {
1862                case CFSDCMD_DAEMONID:
1863                        dcmd_routine = cachefs_io_daemonid;
1864                        break;
1865                case CFSDCMD_STATEGET:
1866                        dcmd_routine = cachefs_io_stateget;
1867                        break;
1868                case CFSDCMD_STATESET:
1869                        dcmd_routine = cachefs_io_stateset;
1870                        break;
1871                case CFSDCMD_XWAIT:
1872                        dcmd_routine = cachefs_io_xwait;
1873                        break;
1874                case CFSDCMD_EXISTS:
1875                        dcmd_routine = cachefs_io_exists;
1876                        break;
1877                case CFSDCMD_LOSTFOUND:
1878                        dcmd_routine = cachefs_io_lostfound;
1879                        break;
1880                case CFSDCMD_GETINFO:
1881                        dcmd_routine = cachefs_io_getinfo;
1882                        break;
1883                case CFSDCMD_CIDTOFID:
1884                        dcmd_routine = cachefs_io_cidtofid;
1885                        break;
1886                case CFSDCMD_GETATTRFID:
1887                        dcmd_routine = cachefs_io_getattrfid;
1888                        break;
1889                case CFSDCMD_GETATTRNAME:
1890                        dcmd_routine = cachefs_io_getattrname;
1891                        break;
1892                case CFSDCMD_GETSTATS:
1893                        dcmd_routine = cachefs_io_getstats;
1894                        break;
1895                case CFSDCMD_ROOTFID:
1896                        dcmd_routine = cachefs_io_rootfid;
1897                        break;
1898                case CFSDCMD_CREATE:
1899                        dcmd_routine = cachefs_io_create;
1900                        break;
1901                case CFSDCMD_REMOVE:
1902                        dcmd_routine = cachefs_io_remove;
1903                        break;
1904                case CFSDCMD_LINK:
1905                        dcmd_routine = cachefs_io_link;
1906                        break;
1907                case CFSDCMD_RENAME:
1908                        dcmd_routine = cachefs_io_rename;
1909                        break;
```



```
1910                case CFSDCMD_MKDIR:
1911                        dcmd_routine = cachefs_io_mkdir;
1912                        break;
1913                case CFSDCMD_RMDIR:
1914                        dcmd_routine = cachefs_io_rmdir;
1915                        break;
1916                case CFSDCMD_SYMLINK:
1917                        dcmd_routine = cachefs_io_symlink;
1918                        break;
1919                case CFSDCMD_SETATTR:
1920                        dcmd_routine = cachefs_io_setattr;
1921                        break;
1922                case CFSDCMD_SETSECATTR:
1923                        dcmd_routine = cachefs_io_setsecattr;
1924                        break;
1925                case CFSDCMD_PUSHBACK:
1926                        dcmd_routine = cachefs_io_pushback;
1927                        break;
1928                default:
1929                        error = ENOTTY;
1930                        break;
1931                }

1933                /* execute the routine */
1934                if (dcmd_routine)
1935                        error = (*dcmd_routine)(vp, dinp, doutp);

1937                /* copy out the result */
1938                if ((error == 0) && doutp)
1939                        error = xcopyout(doutp, STRUCT_FGETP(dcmd, d_rdata),
1940                            outlen);

1942                /* free allocated memory */
1943                if (dinp)
1944                        cachefs_kmem_free(dinp, inlen);
1945                if (doutp)
1946                        cachefs_kmem_free(doutp, outlen);

1948                break;

1950        case _FIOCOD:
1951                if (secpolicy_fs_config(cred, vp->v_vfsp) != 0) {
1952                        error = EPERM;
1953                        break;
1954                }

1956                error = EBUSY;
1957                if (arg) {
1958                        /* non-zero arg means do all filesystems */
1959                        mutex_enter(&cachefs_cachelock);
1960                        for (cachep = cachefs_cachelist; cachep != NULL;
1961                            cachep = cachep->c_next) {
1962                                mutex_enter(&cachep->c_fslistlock);
1963                                for (fscp = cachep->c_fslist;
1964                                    fscp != NULL;
1965                                    fscp = fscp->fs_next) {
1966                                        if (CFS_ISFS_CODCONST(fscp)) {
1967                                                gethrestime(&fscp->fs_cod_time);
1968                                                error = 0;
1969                                        }
1970                                }
1971                                mutex_exit(&cachep->c_fslistlock);
1972                        }
1973                        mutex_exit(&cachefs_cachelock);
1974                } else {
1975                        if (CFS_ISFS_CODCONST(fscp)) {
```

```
1976                                gethrestime(&fscp->fs_cod_time);
1977                                error = 0;
1978                        }
1979                }
1980                break;

1982        case _FIOSTOPCACHE:
1983                error = cachefs_stop_cache(cp);
1984                break;

1986        default:
1987                error = ENOTTY;
1988                break;
1989        }

1991        /* return the result */
1992        return (error);
1993 }

1995 ino64_t
1996 cachefs_fileno_conflict(fscache_t *fscp, ino64_t old)
1997 {
1998        ino64_t new;

2000        ASSERT(MUTEX_HELD(&fscp->fs_fslock));

2002        for (;;) {
2003                fscp->fs_info.fi_localfileno++;
2004                if (fscp->fs_info.fi_localfileno == 0)
2005                        fscp->fs_info.fi_localfileno = 3;
2006                fscp->fs_flags |= CFS_FS_DIRTYINFO;

2008                new = fscp->fs_info.fi_localfileno;
2009                if (! cachefs_fileno_inuse(fscp, new))
2010                        break;
2011        }

2013        cachefs_inum_register(fscp, old, new);
2014        cachefs_inum_register(fscp, new, 0);
2015        return (new);
2016 }

2018 /*ARGSUSED*/
2019 static int
2020 cachefs_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2021        caller_context_t *ct)
2022 {
2023        struct cnode *cp = VTOC(vp);
2024        fscache_t *fscp = C_TO_FSCACHE(cp);
2025        int error = 0;
2026        int held = 0;
2027        int connected = 0;

2029 #ifdef CFSDEBUG
2030        CFS_DEBUG(CFSDEBUG_VOPS)
2031                printf("cachefs_getattr: ENTER vp %p\n", (void *)vp);
2032 #endif

2034        if (getzoneid() != GLOBAL_ZONEID)
2035                return (EPERM);

2037        /* Call backfilesystem getattr if NFSv4 */
2038        if (CFS_ISFS_BACKFS_NFSV4(fscp)) {
2039                error = cachefs_getattr_backfs_nfsv4(vp, vap, flags, cr, ct);
2040                goto out;
2041        }
```



```
2043        /*
2044         * If it has been specified that the return value will
2045         * just be used as a hint, and we are only being asked
2046         * for size, fsid or rdevid, then return the client's
2047         * notion of these values without checking to make sure
2048         * that the attribute cache is up to date.
2049         * The whole point is to avoid an over the wire GETATTR
2050         * call.
2051         */
2052        if (flags & ATTR_HINT) {
2053                if (vap->va_mask ==
2054                    (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
2055                        if (vap->va_mask | AT_SIZE)
2056                                vap->va_size = cp->c_size;
2057                        /*
2058                         * Return the FSID of the cachefs filesystem,
2059                         * not the back filesystem
2060                         */
2061                        if (vap->va_mask | AT_FSID)
2062                                vap->va_fsid = vp->v_vfsp->vfs_dev;
2063                        if (vap->va_mask | AT_RDEV)
2064                                vap->va_rdev = cp->c_attr.va_rdev;
2065                        return (0);
2066                }
2067        }

2069        /*
2070         * Only need to flush pages if asking for the mtime
2071         * and if there any dirty pages.
2072         */
2073        if (vap->va_mask & AT_MTIME) {
2074                /*EMPTY*/
2075 #if 0
2076                /*
2077                 * XXX bob: stolen from nfs code, need to do something similar
2078                 */
2079                rp = VTOR(vp);
2080                if ((rp->r_flags & RDIRTY) || rp->r_iocnt > 0)
2081                        (void) nfs3_putpage(vp, (offset_t)0, 0, 0, cr);
2082 #endif
2083        }

2085        for (;;) {
2086                /* get (or renew) access to the file system */
2087                if (held) {
2088                        cachefs_cd_release(fscp);
2089                        held = 0;
2090                }
2091                error = cachefs_cd_access(fscp, connected, 0);
2092                if (error)
2093                        goto out;
2094                held = 1;

2096                /*
2097                 * If it has been specified that the return value will
2098                 * just be used as a hint, and we are only being asked
2099                 * for size, fsid or rdevid, then return the client's
2100                 * notion of these values without checking to make sure
2101                 * that the attribute cache is up to date.
2102                 * The whole point is to avoid an over the wire GETATTR
2103                 * call.
2104                 */
2105                if (flags & ATTR_HINT) {
2106                        if (vap->va_mask ==
2107                            (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
```

```
2108                                if (vap->va_mask | AT_SIZE)
2109                                        vap->va_size = cp->c_size;
2110                                /*
2111                                 * Return the FSID of the cachefs filesystem,
2112                                 * not the back filesystem
2113                                 */
2114                                if (vap->va_mask | AT_FSID)
2115                                        vap->va_fsid = vp->v_vfsp->vfs_dev;
2116                                if (vap->va_mask | AT_RDEV)
2117                                        vap->va_rdev = cp->c_attr.va_rdev;
2118                                goto out;
2119                        }
2120                }

2122                mutex_enter(&cp->c_statelock);
2123                if ((cp->c_metadata.md_flags & MD_NEEDATTRS) &&
2124                    (fscp->fs_cdconnected != CFS_CD_CONNECTED)) {
2125                        mutex_exit(&cp->c_statelock);
2126                        connected = 1;
2127                        continue;
2128                }

2130                error = CFSOP_CHECK_COBJECT(fscp, cp, 0, cr);
2131                if (CFS_TIMEOUT(fscp, error)) {
2132                        mutex_exit(&cp->c_statelock);
2133                        cachefs_cd_release(fscp);
2134                        held = 0;
2135                        cachefs_cd_timedout(fscp);
2136                        continue;
2137                }
2138                if (error) {
2139                        mutex_exit(&cp->c_statelock);
2140                        break;
2141                }

2143                /* check for fileno conflict */
2144                if ((fscp->fs_inum_size > 0) &&
2145                    ((cp->c_metadata.md_flags & MD_LOCALFILENO) == 0)) {
2146                        ino64_t fakenum;

2148                        mutex_exit(&cp->c_statelock);
2149                        mutex_enter(&fscp->fs_fslock);
2150                        fakenum = cachefs_inum_real2fake(fscp,
2151                            cp->c_attr.va_nodeid);
2152                        if (fakenum == 0) {
2153                                fakenum = cachefs_fileno_conflict(fscp,
2154                                    cp->c_attr.va_nodeid);
2155                        }
2156                        mutex_exit(&fscp->fs_fslock);

2158                        mutex_enter(&cp->c_statelock);
2159                        cp->c_metadata.md_flags |= MD_LOCALFILENO;
2160                        cp->c_metadata.md_localfileno = fakenum;
2161                        cp->c_flags |= CN_UPDATED;
2162                }

2164                /* copy out the attributes */
2165                *vap = cp->c_attr;

2167                /*
2168                 * return the FSID of the cachefs filesystem,
2169                 * not the back filesystem
2170                 */
2171                vap->va_fsid = vp->v_vfsp->vfs_dev;

2173                /* return our idea of the size */
```



```
2174                if (cp->c_size > vap->va_size)
2175                        vap->va_size = cp->c_size;

2177                /* overwrite with our version of fileno and timestamps */
2178                vap->va_nodeid = cp->c_metadata.md_localfileno;
2179                vap->va_mtime = cp->c_metadata.md_localmtime;
2180                vap->va_ctime = cp->c_metadata.md_localctime;

2182                mutex_exit(&cp->c_statelock);
2183                break;
2184        }
2185 out:
2186        if (held)
2187                cachefs_cd_release(fscp);
2188 #ifdef CFS_CD_DEBUG
2189        ASSERT((curthread->t_flag & T_CD_HELD) == 0);
2190 #endif

2192 #ifdef CFSDEBUG
2193        CFS_DEBUG(CFSDEBUG_VOPS)
2194                printf("cachefs_getattr: EXIT error = %d\n", error);
2195 #endif
2196        return (error);
2197 }

2199 /*
2200  * cachefs_getattr_backfs_nfsv4
2201  *
2202  * Call NFSv4 back filesystem to handle the getattr (cachefs
2203  * pass-through support for NFSv4).
2204  */
2205 static int
2206 cachefs_getattr_backfs_nfsv4(vnode_t *vp, vattr_t *vap,
2207     int flags, cred_t *cr, caller_context_t *ct)
2208 {
2209        cnode_t *cp = VTOC(vp);
2210        fscache_t *fscp = C_TO_FSCACHE(cp);
2211        vnode_t *backvp;
2212        int error;

2214        /*
2215         * For NFSv4 pass-through to work, only connected operation
2216         * is supported, the cnode backvp must exist, and cachefs
2217         * optional (eg., disconnectable) flags are turned off. Assert
2218         * these conditions for the getattr operation.
2219         */
2220        CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
2221        CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

2223        /* Call backfs vnode op after extracting backvp */
2224        mutex_enter(&cp->c_statelock);
2225        backvp = cp->c_backvp;
2226        mutex_exit(&cp->c_statelock);

2228        CFS_DPRINT_BACKFS_NFSV4(fscp, ("cachefs_getattr_backfs_nfsv4: cnode %p,"
2229            " backvp %p\n", cp, backvp));
2230        error = VOP_GETATTR(backvp, vap, flags, cr, ct);

2232        /* Update attributes */
2233        cp->c_attr = *vap;

2235        /*
2236         * return the FSID of the cachefs filesystem,
2237         * not the back filesystem
2238         */
2239        vap->va_fsid = vp->v_vfsp->vfs_dev;
```

```
2241         return (error);
2242 }

2244 /*ARGSUSED4*/
2245 static int
2246 cachefs_setattr(
2247         vnode_t *vp,
2248         vattr_t *vap,
2249         int flags,
2250         cred_t *cr,
2251         caller_context_t *ct)
2252 {
2253         cnode_t *cp = VTOC(vp);
2254         fscache_t *fscp = C_TO_FSCACHE(cp);
2255         int error;
2256         int connected;
2257         int held = 0;

2259         if (getzoneid() != GLOBAL_ZONEID)
2260                 return (EPERM);

2262         /*
2263          * Cachefs only provides pass-through support for NFSv4,
2264          * and all vnode operations are passed through to the
2265          * back file system. For NFSv4 pass-through to work, only
2266          * connected operation is supported, the cnode backvp must
2267          * exist, and cachefs optional (eg., disconnectable) flags
2268          * are turned off. Assert these conditions to ensure that
2269          * the backfilesystem is called for the setattr operation.
2270          */
2271         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
2272         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

2274         connected = 0;
2275         for (;;) {
2276                 /* drop hold on file system */
2277                 if (held) {
2278                         /* Won't loop with NFSv4 connected behavior */
2279                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
2280                         cachefs_cd_release(fscp);
2281                         held = 0;
2282                 }

2284                 /* acquire access to the file system */
2285                 error = cachefs_cd_access(fscp, connected, 1);
2286                 if (error)
2287                         break;
2288                 held = 1;

2290                 /* perform the setattr */
2291                 error = cachefs_setattr_common(vp, vap, flags, cr, ct);
2292                 if (error) {
2293                         /* if connected */
2294                         if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
2295                                 if (CFS_TIMEOUT(fscp, error)) {
2296                                         cachefs_cd_release(fscp);
2297                                         held = 0;
2298                                         cachefs_cd_timedout(fscp);
2299                                         connected = 0;
2300                                         continue;
2301                                 }
2302                         }

2304                         /* else must be disconnected */
2305                         else {
```



```
2306                                 if (CFS_TIMEOUT(fscp, error)) {
2307                                         connected = 1;
2308                                         continue;
2309                                 }
2310                         }
2311                 }
2312                 break;
2313         }

2315         if (held) {
2316                 cachefs_cd_release(fscp);
2317         }
2318 #ifdef CFS_CD_DEBUG
2319         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
2320 #endif
2321         return (error);
2322 }

2324 static int
2325 cachefs_setattr_common(
2326         vnode_t *vp,
2327         vattr_t *vap,
2328         int flags,
2329         cred_t *cr,
2330         caller_context_t *ct)
2331 {
2332         cnode_t *cp = VTOC(vp);
2333         fscache_t *fscp = C_TO_FSCACHE(cp);
2334         cachefscache_t *cachep = fscp->fs_cache;
2335         uint_t mask = vap->va_mask;
2336         int error = 0;
2337         uint_t bcnt;

2339         /* Cannot set these attributes. */
2340         if (mask & AT_NOSET)
2341                 return (EINVAL);

2343         /*
2344          * Truncate file.  Must have write permission and not be a directory.
2345          */
2346         if (mask & AT_SIZE) {
2347                 if (vp->v_type == VDIR) {
2348                         if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_TRUNCATE))
2349                                 cachefs_log_truncate(cachep, EISDIR,
2350                                     fscp->fs_cfsvfsp,
2351                                     &cp->c_metadata.md_cookie,
2352                                     cp->c_id.cid_fileno,
2353                                     crgetuid(cr), vap->va_size);
2354                         return (EISDIR);
2355                 }
2356         }

2358         /*
2359          * Gotta deal with one special case here, where we're setting the
2360          * size of the file. First, we zero out part of the page after the
2361          * new size of the file. Then we toss (not write) all pages after
2362          * page in which the new offset occurs. Note that the NULL passed
2363          * in instead of a putapage() fn parameter is correct, since
2364          * no dirty pages will be found (B_TRUNC | B_INVAL).
2365          */

2367         rw_enter(&cp->c_rwlock, RW_WRITER);

2369         /* sync dirty pages */
2370         if (!CFS_ISFS_BACKFS_NFSV4(fscp)) {
2371                 error = cachefs_putpage_common(vp, (offset_t)0, 0, 0, cr);
```

```
2372                 if (error == EINTR)
2373                         goto out;
2374         }
2375         error = 0;

2377         /* if connected */
2378         if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
2379                 error = cachefs_setattr_connected(vp, vap, flags, cr, ct);
2380         }
2381         /* else must be disconnected */
2382         else {
2383                 error = cachefs_setattr_disconnected(vp, vap, flags, cr, ct);
2384         }
2385         if (error)
2386                 goto out;

2388         /*
2389          * If the file size has been changed then
2390          * toss whole pages beyond the end of the file and zero
2391          * the portion of the last page that is beyond the end of the file.
2392          */
2393         if (mask & AT_SIZE && !CFS_ISFS_BACKFS_NFSV4(fscp)) {
2394                 bcnt = (uint_t)(cp->c_size & PAGEOFFSET);
2395                 if (bcnt)
2396                         pvn_vpzero(vp, cp->c_size, PAGESIZE - bcnt);
2397                 (void) pvn_vplist_dirty(vp, cp->c_size, cachefs_push,
2398                     B_TRUNC | B_INVAL, cr);
2399         }

2401 out:
2402         rw_exit(&cp->c_rwlock);

2404         if ((mask & AT_SIZE) &&
2405             (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_TRUNCATE)))
2406                 cachefs_log_truncate(cachep, error, fscp->fs_cfsvfsp,
2407                     &cp->c_metadata.md_cookie, cp->c_id.cid_fileno,
2408                     crgetuid(cr), vap->va_size);

2410         return (error);
2411 }

2413 static int
2414 cachefs_setattr_connected(
2415         vnode_t *vp,
2416         vattr_t *vap,
2417         int flags,
2418         cred_t *cr,
2419         caller_context_t *ct)
2420 {
2421         cnode_t *cp = VTOC(vp);
2422         fscache_t *fscp = C_TO_FSCACHE(cp);
2423         uint_t mask = vap->va_mask;
2424         int error = 0;
2425         int setsize;

2427         mutex_enter(&cp->c_statelock);

2429         if (cp->c_backvp == NULL) {
2430                 error = cachefs_getbackvp(fscp, cp);
2431                 if (error)
2432                         goto out;
2433         }

2435         error = CFSOP_CHECK_COBJECT(fscp, cp, 0, cr);
2436         if (error)
2437                 goto out;
```



```
2439         CFS_DPRINT_BACKFS_NFSV4(fscp, ("cachefs_setattr (nfsv4): cnode %p, "
2440             "backvp %p\n", cp, cp->c_backvp));
2441         error = VOP_SETATTR(cp->c_backvp, vap, flags, cr, ct);
2442         if (error) {
2443                 goto out;
2444         }

2446         /* if the size of the file is being changed */
2447         if (mask & AT_SIZE) {
2448                 cp->c_size = vap->va_size;
2449                 error = 0;
2450                 setsize = 0;

2452                 /* see if okay to try to set the file size */
2453                 if (((cp->c_flags & CN_NOCACHE) == 0) &&
2454                     CFS_ISFS_NONSHARED(fscp)) {
2455                         /* okay to set size if file is populated */
2456                         if (cp->c_metadata.md_flags & MD_POPULATED)
2457                                 setsize = 1;

2459                         /*
2460                          * Okay to set size if front file exists and setting
2461                          * file size to zero.
2462                          */
2463                         if ((cp->c_metadata.md_flags & MD_FILE) &&
2464                             (vap->va_size == 0))
2465                                 setsize = 1;
2466                 }

2468                 /* if okay to try to set the file size */
2469                 if (setsize) {
2470                         error = 0;
2471                         if (cp->c_frontvp == NULL)
2472                                 error = cachefs_getfrontfile(cp);
2473                         if (error == 0)
2474                                 error = cachefs_frontfile_size(cp, cp->c_size);
2475                 } else if (cp->c_metadata.md_flags & MD_FILE) {
2476                         /* make sure file gets nocached */
2477                         error = EEXIST;
2478                 }

2480                 /* if we have to nocache the file */
2481                 if (error) {
2482                         if ((cp->c_flags & CN_NOCACHE) == 0 &&
2483                             !CFS_ISFS_BACKFS_NFSV4(fscp))
2484                                 cachefs_nocache(cp);
2485                         error = 0;
2486                 }
2487         }

2489         cp->c_flags |= CN_UPDATED;

2491         /* XXX bob: given what modify_cobject does this seems unnecessary */
2492         cp->c_attr.va_mask = AT_ALL;
2493         error = VOP_GETATTR(cp->c_backvp, &cp->c_attr, 0, cr, ct);
2494         if (error)
2495                 goto out;

2497         cp->c_attr.va_size = MAX(cp->c_attr.va_size, cp->c_size);
2498         cp->c_size = cp->c_attr.va_size;

2500         CFSOP_MODIFY_COBJECT(fscp, cp, cr);
2501 out:
2502         mutex_exit(&cp->c_statelock);
2503         return (error);
```

```
2504 }
2506 /*
2507  * perform the setattr on the local file system
2508  */
2509 /*ARGSUSED4*/
2510 static int
2511 cachefs_setattr_disconnected(
2512         vnode_t *vp,
2513         vattr_t *vap,
2514         int flags,
2515         cred_t *cr,
2516         caller_context_t *ct)
2517 {
2518         cnode_t *cp = VTOC(vp);
2519         fscache_t *fscp = C_TO_FSCACHE(cp);
2520         int mask;
2521         int error;
2522         int newfile;
2523         off_t commit = 0;
2525         if (CFS_ISFS_WRITE_AROUND(fscp))
2526                 return (ETIMEDOUT);
2528         /* if we do not have good attributes */
2529         if (cp->c_metadata.md_flags & MD_NEEDATTRS)
2530                 return (ETIMEDOUT);
2532         /* primary concern is to keep this routine as much like ufs_setattr */
2534         mutex_enter(&cp->c_statelock);
2536         error = secpolicy_vnode_setattr(cr, vp, vap, &cp->c_attr, flags,
2537             cachefs_access_local, cp);
2539         if (error)
2540                 goto out;
2542         mask = vap->va_mask;
2544         /* if changing the size of the file */
2545         if (mask & AT_SIZE) {
2546                 if (vp->v_type == VDIR) {
2547                         error = EISDIR;
2548                         goto out;
2549                 }
2551                 if (vp->v_type == VFIFO) {
2552                         error = 0;
2553                         goto out;
2554                 }
2556                 if ((vp->v_type != VREG) &&
2557                     !((vp->v_type == VLNK) && (vap->va_size == 0))) {
2558                         error = EINVAL;
2559                         goto out;
2560                 }
2562                 if (vap->va_size > fscp->fs_offmax) {
2563                         error = EFBIG;
2564                         goto out;
2565                 }
2567                 /* if the file is not populated and we are not truncating it */
2568                 if (((cp->c_metadata.md_flags & MD_POPULATED) == 0) &&
2569                     (vap->va_size != 0)) {
```



```
2570                         error = ETIMEDOUT;
2571                         goto out;
2572                 }
2574                 if ((cp->c_metadata.md_flags & MD_MAPPING) == 0) {
2575                         error = cachefs_dlog_cidmap(fscp);
2576                         if (error) {
2577                                 error = ENOSPC;
2578                                 goto out;
2579                         }
2580                         cp->c_metadata.md_flags |= MD_MAPPING;
2581                 }
2583                 /* log the operation */
2584                 commit = cachefs_dlog_setattr(fscp, vap, flags, cp, cr);
2585                 if (commit == 0) {
2586                         error = ENOSPC;
2587                         goto out;
2588                 }
2589                 cp->c_flags &= ~CN_NOCACHE;
2591                 /* special case truncating fast sym links */
2592                 if ((vp->v_type == VLNK) &&
2593                     (cp->c_metadata.md_flags & MD_FASTSYMLNK)) {
2594                         /* XXX how can we get here */
2595                         /* XXX should update mtime */
2596                         cp->c_size = 0;
2597                         error = 0;
2598                         goto out;
2599                 }
2601                 /* get the front file, this may create one */
2602                 newfile = (cp->c_metadata.md_flags & MD_FILE) ? 0 : 1;
2603                 if (cp->c_frontvp == NULL) {
2604                         error = cachefs_getfrontfile(cp);
2605                         if (error)
2606                                 goto out;
2607                 }
2608                 ASSERT(cp->c_frontvp);
2609                 if (newfile && (cp->c_flags & CN_UPDATED)) {
2610                         /* allocate space for the metadata */
2611                         ASSERT((cp->c_flags & CN_ALLOC_PENDING) == 0);
2612                         ASSERT((cp->c_filegrp->fg_flags & CFS_FG_ALLOC_ATTR)
2613                             == 0);
2614                         error = filegrp_write_metadata(cp->c_filegrp,
2615                             &cp->c_id, &cp->c_metadata);
2616                         if (error)
2617                                 goto out;
2618                 }
2620                 /* change the size of the front file */
2621                 error = cachefs_frontfile_size(cp, vap->va_size);
2622                 if (error)
2623                         goto out;
2624                 cp->c_attr.va_size = cp->c_size = vap->va_size;
2625                 gethrestime(&cp->c_metadata.md_localmtime);
2626                 cp->c_metadata.md_flags |= MD_POPULATED | MD_LOCALMTIME;
2627                 cachefs_modified(cp);
2628                 cp->c_flags |= CN_UPDATED;
2629         }
2631         if (mask & AT_MODE) {
2632                 /* mark as modified */
2633                 if (cachefs_modified_alloc(cp)) {
2634                         error = ENOSPC;
2635                         goto out;
```

```
2636                }

2638                if ((cp->c_metadata.md_flags & MD_MAPPING) == 0) {
2639                        error = cachefs_dlog_cidmap(fscp);
2640                        if (error) {
2641                                error = ENOSPC;
2642                                goto out;
2643                        }
2644                        cp->c_metadata.md_flags |= MD_MAPPING;
2645                }

2647                /* log the operation if not already logged */
2648                if (commit == 0) {
2649                        commit = cachefs_dlog_setattr(fscp, vap, flags, cp, cr);
2650                        if (commit == 0) {
2651                                error = ENOSPC;
2652                                goto out;
2653                        }
2654                }

2656                cp->c_attr.va_mode &= S_IFMT;
2657                cp->c_attr.va_mode |= vap->va_mode & ~S_IFMT;
2658                gethrestime(&cp->c_metadata.md_localctime);
2659                cp->c_metadata.md_flags |= MD_LOCALCTIME;
2660                cp->c_flags |= CN_UPDATED;
2661        }

2663        if (mask & (AT_UID|AT_GID)) {

2665                /* mark as modified */
2666                if (cachefs_modified_alloc(cp)) {
2667                        error = ENOSPC;
2668                        goto out;
2669                }

2671                if ((cp->c_metadata.md_flags & MD_MAPPING) == 0) {
2672                        error = cachefs_dlog_cidmap(fscp);
2673                        if (error) {
2674                                error = ENOSPC;
2675                                goto out;
2676                        }
2677                        cp->c_metadata.md_flags |= MD_MAPPING;
2678                }

2680                /* log the operation if not already logged */
2681                if (commit == 0) {
2682                        commit = cachefs_dlog_setattr(fscp, vap, flags, cp, cr);
2683                        if (commit == 0) {
2684                                error = ENOSPC;
2685                                goto out;
2686                        }
2687                }

2689                if (mask & AT_UID)
2690                        cp->c_attr.va_uid = vap->va_uid;

2692                if (mask & AT_GID)
2693                        cp->c_attr.va_gid = vap->va_gid;
2694                gethrestime(&cp->c_metadata.md_localctime);
2695                cp->c_metadata.md_flags |= MD_LOCALCTIME;
2696                cp->c_flags |= CN_UPDATED;
2697        }


2700        if (mask & (AT_MTIME|AT_ATIME)) {
2701                /* mark as modified */
```



```
2702                if (cachefs_modified_alloc(cp)) {
2703                        error = ENOSPC;
2704                        goto out;
2705                }

2707                if ((cp->c_metadata.md_flags & MD_MAPPING) == 0) {
2708                        error = cachefs_dlog_cidmap(fscp);
2709                        if (error) {
2710                                error = ENOSPC;
2711                                goto out;
2712                        }
2713                        cp->c_metadata.md_flags |= MD_MAPPING;
2714                }

2716                /* log the operation if not already logged */
2717                if (commit == 0) {
2718                        commit = cachefs_dlog_setattr(fscp, vap, flags, cp, cr);
2719                        if (commit == 0) {
2720                                error = ENOSPC;
2721                                goto out;
2722                        }
2723                }

2725                if (mask & AT_MTIME) {
2726                        cp->c_metadata.md_localmtime = vap->va_mtime;
2727                        cp->c_metadata.md_flags |= MD_LOCALMTIME;
2728                }
2729                if (mask & AT_ATIME)
2730                        cp->c_attr.va_atime = vap->va_atime;
2731                gethrestime(&cp->c_metadata.md_localctime);
2732                cp->c_metadata.md_flags |= MD_LOCALCTIME;
2733                cp->c_flags |= CN_UPDATED;
2734        }

2736 out:
2737        mutex_exit(&cp->c_statelock);

2739        /* commit the log entry */
2740        if (commit) {
2741                if (cachefs_dlog_commit(fscp, commit, error)) {
2742                        /*EMPTY*/
2743                        /* XXX bob: fix on panic */
2744                }
2745        }
2746        return (error);
2747 }

2749 /* ARGSUSED */
2750 static int
2751 cachefs_access(vnode_t *vp, int mode, int flags, cred_t *cr,
2752        caller_context_t *ct)
2753 {
2754        cnode_t *cp = VTOC(vp);
2755        fscache_t *fscp = C_TO_FSCACHE(cp);
2756        int error;
2757        int held = 0;
2758        int connected = 0;

2760 #ifdef CFSDEBUG
2761        CFS_DEBUG(CFSDEBUG_VOPS)
2762                printf("cachefs_access: ENTER vp %p\n", (void *)vp);
2763 #endif
2764        if (getzoneid() != GLOBAL_ZONEID) {
2765                error = EPERM;
2766                goto out;
2767        }
```

```
2769         /*
2770          * Cachefs only provides pass-through support for NFSv4,
2771          * and all vnode operations are passed through to the
2772          * back file system. For NFSv4 pass-through to work, only
2773          * connected operation is supported, the cnode backvp must
2774          * exist, and cachefs optional (eg., disconnectable) flags
2775          * are turned off. Assert these conditions to ensure that
2776          * the backfilesystem is called for the access operation.
2777          */
2778         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
2779         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

2781         for (;;) {
2782                 /* get (or renew) access to the file system */
2783                 if (held) {
2784                         /* Won't loop with NFSv4 connected behavior */
2785                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
2786                         cachefs_cd_release(fscp);
2787                         held = 0;
2788                 }
2789                 error = cachefs_cd_access(fscp, connected, 0);
2790                 if (error)
2791                         break;
2792                 held = 1;

2794                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
2795                         error = cachefs_access_connected(vp, mode, flags,
2796                             cr);
2797                         if (CFS_TIMEOUT(fscp, error)) {
2798                                 cachefs_cd_release(fscp);
2799                                 held = 0;
2800                                 cachefs_cd_timedout(fscp);
2801                                 connected = 0;
2802                                 continue;
2803                         }
2804                 } else {
2805                         mutex_enter(&cp->c_statelock);
2806                         error = cachefs_access_local(cp, mode, cr);
2807                         mutex_exit(&cp->c_statelock);
2808                         if (CFS_TIMEOUT(fscp, error)) {
2809                                 if (cachefs_cd_access_miss(fscp)) {
2810                                         mutex_enter(&cp->c_statelock);
2811                                         if (cp->c_backvp == NULL) {
2812                                                 (void) cachefs_getbackvp(fscp,
2813                                                     cp);
2814                                         }
2815                                         mutex_exit(&cp->c_statelock);
2816                                         error = cachefs_access_connected(vp,
2817                                             mode, flags, cr);
2818                                         if (!CFS_TIMEOUT(fscp, error))
2819                                                 break;
2820                                         delay(5*hz);
2821                                         connected = 0;
2822                                         continue;
2823                                 }
2824                                 connected = 1;
2825                                 continue;
2826                         }
2827                 }
2828                 break;
2829         }
2830         if (held)
2831                 cachefs_cd_release(fscp);
2832 #ifdef CFS_CD_DEBUG
2833         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
```



```
2834 #endif
2835 out:
2836 #ifdef CFSDEBUG
2837         CFS_DEBUG(CFSDEBUG_VOPS)
2838                 printf("cachefs_access: EXIT error = %d\n", error);
2839 #endif
2840         return (error);
2841 }

2843 static int
2844 cachefs_access_connected(struct vnode *vp, int mode, int flags, cred_t *cr)
2845 {
2846         cnode_t *cp = VTOC(vp);
2847         fscache_t *fscp = C_TO_FSCACHE(cp);
2848         int error = 0;

2850         mutex_enter(&cp->c_statelock);

2852         /* Make sure the cnode attrs are valid first. */
2853         error = CFSOP_CHECK_COBJECT(fscp, cp, 0, cr);
2854         if (error)
2855                 goto out;

2857         /* see if can do a local file system check */
2858         if ((fscp->fs_info.fi_mntflags & CFS_ACCESS_BACKFS) == 0 &&
2859             !CFS_ISFS_BACKFS_NFSV4(fscp)) {
2860                 error = cachefs_access_local(cp, mode, cr);
2861                 goto out;
2862         }

2864         /* else do a remote file system check */
2865         else {
2866                 if (cp->c_backvp == NULL) {
2867                         error = cachefs_getbackvp(fscp, cp);
2868                         if (error)
2869                                 goto out;
2870                 }

2872                 CFS_DPRINT_BACKFS_NFSV4(fscp,
2873                     ("cachefs_access (nfsv4): cnode %p, backvp %p\n",
2874                     cp, cp->c_backvp));
2875                 error = VOP_ACCESS(cp->c_backvp, mode, flags, cr, NULL);

2877                 /*
2878                  * even though we don't 'need' the ACL to do access
2879                  * via the backvp, we should cache it here to make our
2880                  * behavior more reasonable if we go disconnected.
2881                  */

2883                 if (((fscp->fs_info.fi_mntflags & CFS_NOACL) == 0) &&
2884                     (cachefs_vtype_aclok(vp)) &&
2885                     ((cp->c_flags & CN_NOCACHE) == 0) &&
2886                     (!CFS_ISFS_BACKFS_NFSV4(fscp)) &&
2887                     ((cp->c_metadata.md_flags & MD_ACL) == 0))
2888                         (void) cachefs_cacheacl(cp, NULL);
2889         }
2890 out:
2891         /*
2892          * If NFS returned ESTALE, mark this cnode as stale, so that
2893          * the vn_open retry will read the file anew from backfs
2894          */
2895         if (error == ESTALE)
2896                 cachefs_cnode_stale(cp);

2898         mutex_exit(&cp->c_statelock);
2899         return (error);
```

```
}

/*
 * CFS has a fastsymlink scheme. If the size of the link is < C_FSL_SIZE, then
 * the link is placed in the metadata itself (no front file is allocated).
 */
/*ARGSUSED*/
static int
cachefs_readlink(vnode_t *vp, uio_t *uiop, cred_t *cr, caller_context_t *ct)
{
        int error = 0;
        cnode_t *cp = VTOC(vp);
        fscache_t *fscp = C_TO_FSCACHE(cp);
        cachefscache_t *cachep = fscp->fs_cache;
        int held = 0;
        int connected = 0;

        if (getzoneid() != GLOBAL_ZONEID)
                return (EPERM);

        if (vp->v_type != VLNK)
                return (EINVAL);

        /*
         * Cachefs only provides pass-through support for NFSv4,
         * and all vnode operations are passed through to the
         * back file system. For NFSv4 pass-through to work, only
         * connected operation is supported, the cnode backvp must
         * exist, and cachefs optional (eg., disconnectable) flags
         * are turned off. Assert these conditions to ensure that
         * the backfilesystem is called for the readlink operation.
         */
        CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
        CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

        for (;;) {
                /* get (or renew) access to the file system */
                if (held) {
                        /* Won't loop with NFSv4 connected behavior */
                        ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
                        cachefs_cd_release(fscp);
                        held = 0;
                }
                error = cachefs_cd_access(fscp, connected, 0);
                if (error)
                        break;
                held = 1;

                if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
                        /*
                         * since readlink_connected will call stuffsymlink
                         * on success, have to serialize access
                         */
                        if (!rw_tryenter(&cp->c_rwlock, RW_WRITER)) {
                                cachefs_cd_release(fscp);
                                rw_enter(&cp->c_rwlock, RW_WRITER);
                                error = cachefs_cd_access(fscp, connected, 0);
                                if (error) {
                                        held = 0;
                                        rw_exit(&cp->c_rwlock);
                                        break;
                                }
                        }
                        error = cachefs_readlink_connected(vp, uiop, cr);
                        rw_exit(&cp->c_rwlock);
                        if (CFS_TIMEOUT(fscp, error)) {
```



```
                                cachefs_cd_release(fscp);
                                held = 0;
                                cachefs_cd_timedout(fscp);
                                connected = 0;
                                continue;
                        }
                } else {
                        error = cachefs_readlink_disconnected(vp, uiop);
                        if (CFS_TIMEOUT(fscp, error)) {
                                if (cachefs_cd_access_miss(fscp)) {
                                        /* as above */
                                        if (!rw_tryenter(&cp->c_rwlock,
                                            RW_WRITER)) {
                                                cachefs_cd_release(fscp);
                                                rw_enter(&cp->c_rwlock,
                                                    RW_WRITER);
                                                error = cachefs_cd_access(fscp,
                                                    connected, 0);
                                                if (error) {
                                                        held = 0;
                                                        rw_exit(&cp->c_rwlock);
                                                        break;
                                                }
                                        }
                                        error = cachefs_readlink_connected(vp,
                                            uiop, cr);
                                        rw_exit(&cp->c_rwlock);
                                        if (!CFS_TIMEOUT(fscp, error))
                                                break;
                                        delay(5*hz);
                                        connected = 0;
                                        continue;
                                }
                                connected = 1;
                                continue;
                        }
                }
                break;
        }
        if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_READLINK))
                cachefs_log_readlink(cachep, error, fscp->fs_cfsvfsp,
                    &cp->c_metadata.md_cookie, cp->c_id.cid_fileno,
                    crgetuid(cr), cp->c_size);

        if (held)
                cachefs_cd_release(fscp);
#ifdef CFS_CD_DEBUG
        ASSERT((curthread->t_flag & T_CD_HELD) == 0);
#endif

        /*
         * The over the wire error for attempting to readlink something
         * other than a symbolic link is ENXIO.  However, we need to
         * return EINVAL instead of ENXIO, so we map it here.
         */
        return (error == ENXIO ? EINVAL : error);
}

static int
cachefs_readlink_connected(vnode_t *vp, uio_t *uiop, cred_t *cr)
{
        int error;
        cnode_t *cp = VTOC(vp);
        fscache_t *fscp = C_TO_FSCACHE(cp);
        caddr_t buf;
        int buflen;
```

```
3032         int readcache = 0;

3034         mutex_enter(&cp->c_statelock);

3036         error = CFSOP_CHECK_COBJECT(fscp, cp, 0, cr);
3037         if (error)
3038                 goto out;

3040         /* if the sym link is cached as a fast sym link */
3041         if (cp->c_metadata.md_flags & MD_FASTSYMLNK) {
3042                 ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
3043                 error = uiomove(cp->c_metadata.md_allocinfo,
3044                     MIN(cp->c_size, uiop->uio_resid), UIO_READ, uiop);
3045 #ifdef CFSDEBUG
3046                 readcache = 1;
3047                 goto out;
3048 #else /* CFSDEBUG */
3049                 /* XXX KLUDGE! correct for insidious 0-len symlink */
3050                 if (cp->c_size != 0) {
3051                         readcache = 1;
3052                         goto out;
3053                 }
3054 #endif /* CFSDEBUG */
3055         }

3057         /* if the sym link is cached in a front file */
3058         if (cp->c_metadata.md_flags & MD_POPULATED) {
3059                 ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
3060                 ASSERT(cp->c_metadata.md_flags & MD_FILE);
3061                 if (cp->c_frontvp == NULL) {
3062                         (void) cachefs_getfrontfile(cp);
3063                 }
3064                 if (cp->c_metadata.md_flags & MD_POPULATED) {
3065                         /* read symlink data from frontfile */
3066                         uiop->uio_offset = 0;
3067                         (void) VOP_RWLOCK(cp->c_frontvp,
3068                             V_WRITELOCK_FALSE, NULL);
3069                         error = VOP_READ(cp->c_frontvp, uiop, 0, kcred, NULL);
3070                         VOP_RWUNLOCK(cp->c_frontvp, V_WRITELOCK_FALSE, NULL);

3072                         /* XXX KLUDGE! correct for insidious 0-len symlink */
3073                         if (cp->c_size != 0) {
3074                                 readcache = 1;
3075                                 goto out;
3076                         }
3077                 }
3078         }

3080         /* get the sym link contents from the back fs */
3081         error = cachefs_readlink_back(cp, cr, &buf, &buflen);
3082         if (error)
3083                 goto out;

3085         /* copy the contents out to the user */
3086         error = uiomove(buf, MIN(buflen, uiop->uio_resid), UIO_READ, uiop);

3088         /*
3089          * try to cache the sym link, note that its a noop if NOCACHE is set
3090          * or if NFSv4 pass-through is enabled.
3091          */
3092         if (cachefs_stuffsymlink(cp, buf, buflen)) {
3093                 cachefs_nocache(cp);
3094         }

3096         cachefs_kmem_free(buf, MAXPATHLEN);
```



```
3098 out:
3099         mutex_exit(&cp->c_statelock);
3100         if (error == 0) {
3101                 if (readcache)
3102                         fscp->fs_stats.st_hits++;
3103                 else
3104                         fscp->fs_stats.st_misses++;
3105         }
3106         return (error);
3107 }

3109 static int
3110 cachefs_readlink_disconnected(vnode_t *vp, uio_t *uiop)
3111 {
3112         int error;
3113         cnode_t *cp = VTOC(vp);
3114         fscache_t *fscp = C_TO_FSCACHE(cp);
3115         int readcache = 0;

3117         mutex_enter(&cp->c_statelock);

3119         /* if the sym link is cached as a fast sym link */
3120         if (cp->c_metadata.md_flags & MD_FASTSYMLNK) {
3121                 error = uiomove(cp->c_metadata.md_allocinfo,
3122                     MIN(cp->c_size, uiop->uio_resid), UIO_READ, uiop);
3123                 readcache = 1;
3124                 goto out;
3125         }

3127         /* if the sym link is cached in a front file */
3128         if (cp->c_metadata.md_flags & MD_POPULATED) {
3129                 ASSERT(cp->c_metadata.md_flags & MD_FILE);
3130                 if (cp->c_frontvp == NULL) {
3131                         (void) cachefs_getfrontfile(cp);
3132                 }
3133                 if (cp->c_metadata.md_flags & MD_POPULATED) {
3134                         /* read symlink data from frontfile */
3135                         uiop->uio_offset = 0;
3136                         (void) VOP_RWLOCK(cp->c_frontvp,
3137                             V_WRITELOCK_FALSE, NULL);
3138                         error = VOP_READ(cp->c_frontvp, uiop, 0, kcred, NULL);
3139                         VOP_RWUNLOCK(cp->c_frontvp, V_WRITELOCK_FALSE, NULL);
3140                         readcache = 1;
3141                         goto out;
3142                 }
3143         }
3144         error = ETIMEDOUT;

3146 out:
3147         mutex_exit(&cp->c_statelock);
3148         if (error == 0) {
3149                 if (readcache)
3150                         fscp->fs_stats.st_hits++;
3151                 else
3152                         fscp->fs_stats.st_misses++;
3153         }
3154         return (error);
3155 }

3157 /*ARGSUSED*/
3158 static int
3159 cachefs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
3160 {
3161         cnode_t *cp = VTOC(vp);
3162         int error = 0;
3163         fscache_t *fscp = C_TO_FSCACHE(cp);
```

```
3164         int held = 0;
3165         int connected = 0;

3167 #ifdef CFSDEBUG
3168         CFS_DEBUG(CFSDEBUG_VOPS)
3169                 printf("cachefs_fsync: ENTER vp %p\n", (void *)vp);
3170 #endif

3172         if (getzoneid() != GLOBAL_ZONEID) {
3173                 error = EPERM;
3174                 goto out;
3175         }

3177         if (fscp->fs_backvfsp && fscp->fs_backvfsp->vfs_flag & VFS_RDONLY)
3178                 goto out;

3180         /*
3181          * Cachefs only provides pass-through support for NFSv4,
3182          * and all vnode operations are passed through to the
3183          * back file system. For NFSv4 pass-through to work, only
3184          * connected operation is supported, the cnode backvp must
3185          * exist, and cachefs optional (eg., disconnectable) flags
3186          * are turned off. Assert these conditions to ensure that
3187          * the backfilesystem is called for the fsync operation.
3188          */
3189         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
3190         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

3192         for (;;) {
3193                 /* get (or renew) access to the file system */
3194                 if (held) {
3195                         /* Won't loop with NFSv4 connected behavior */
3196                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
3197                         cachefs_cd_release(fscp);
3198                         held = 0;
3199                 }
3200                 error = cachefs_cd_access(fscp, connected, 1);
3201                 if (error)
3202                         break;
3203                 held = 1;
3204                 connected = 0;

3206                 /* if a regular file, write out the pages */
3207                 if ((vp->v_type == VREG) && vn_has_cached_data(vp) &&
3208                     !CFS_ISFS_BACKFS_NFSV4(fscp)) {
3209                         error = cachefs_putpage_common(vp, (offset_t)0,
3210                             0, 0, cr);
3211                         if (CFS_TIMEOUT(fscp, error)) {
3212                                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
3213                                         cachefs_cd_release(fscp);
3214                                         held = 0;
3215                                         cachefs_cd_timedout(fscp);
3216                                         continue;
3217                                 } else {
3218                                         connected = 1;
3219                                         continue;
3220                                 }
3221                         }

3223                         /* if no space left in cache, wait until connected */
3224                         if ((error == ENOSPC) &&
3225                             (fscp->fs_cdconnected != CFS_CD_CONNECTED)) {
3226                                 connected = 1;
3227                                 continue;
3228                         }
```



```
3230                         /* clear the cnode error if putpage worked */
3231                         if ((error == 0) && cp->c_error) {
3232                                 mutex_enter(&cp->c_statelock);
3233                                 cp->c_error = 0;
3234                                 mutex_exit(&cp->c_statelock);
3235                         }

3237                         if (error)
3238                                 break;
3239                 }

3241                 /* if connected, sync the backvp */
3242                 if ((fscp->fs_cdconnected == CFS_CD_CONNECTED) &&
3243                     cp->c_backvp) {
3244                         mutex_enter(&cp->c_statelock);
3245                         if (cp->c_backvp) {
3246                                 CFS_DPRINT_BACKFS_NFSV4(fscp,
3247                                     ("cachefs_fsync (nfsv4): cnode %p, "
3248                                     "backvp %p\n", cp, cp->c_backvp));
3249                                 error = VOP_FSYNC(cp->c_backvp, syncflag, cr,
3250                                     ct);
3251                                 if (CFS_TIMEOUT(fscp, error)) {
3252                                         mutex_exit(&cp->c_statelock);
3253                                         cachefs_cd_release(fscp);
3254                                         held = 0;
3255                                         cachefs_cd_timedout(fscp);
3256                                         continue;
3257                                 } else if (error && (error != EINTR))
3258                                         cp->c_error = error;
3259                         }
3260                         mutex_exit(&cp->c_statelock);
3261                 }

3263                 /* sync the metadata and the front file to the front fs */
3264                 if (!CFS_ISFS_BACKFS_NFSV4(fscp)) {
3265                         error = cachefs_sync_metadata(cp);
3266                         if (error &&
3267                             (fscp->fs_cdconnected == CFS_CD_CONNECTED))
3268                                 error = 0;
3269                 }
3270                 break;
3271         }

3273         if (error == 0)
3274                 error = cp->c_error;

3276         if (held)
3277                 cachefs_cd_release(fscp);

3279 out:
3280 #ifdef CFS_CD_DEBUG
3281         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
3282 #endif

3284 #ifdef CFSDEBUG
3285         CFS_DEBUG(CFSDEBUG_VOPS)
3286                 printf("cachefs_fsync: EXIT vp %p\n", (void *)vp);
3287 #endif
3288         return (error);
3289 }

3291 /*
3292  * Called from cachefs_inactive(), to make sure all the data goes out to disk.
3293  */
3294 int
3295 cachefs_sync_metadata(cnode_t *cp)
```

```
{
        int error = 0;
        struct filegrp *fgp;
        struct vattr va;
        fscache_t *fscp = C_TO_FSCACHE(cp);

#ifdef CFSDEBUG
        CFS_DEBUG(CFSDEBUG_VOPS)
                printf("c_sync_metadata: ENTER cp %p cflag %x\n",
                    (void *)cp, cp->c_flags);
#endif

        mutex_enter(&cp->c_statelock);
        if ((cp->c_flags & CN_UPDATED) == 0)
                goto out;
        if (cp->c_flags & (CN_STALE | CN_DESTROY))
                goto out;
        fgp = cp->c_filegrp;
        if ((fgp->fg_flags & CFS_FG_WRITE) == 0)
                goto out;
        if (CFS_ISFS_BACKFS_NFSV4(fscp))
                goto out;

        if (fgp->fg_flags & CFS_FG_ALLOC_ATTR) {
                mutex_exit(&cp->c_statelock);
                error = filegrp_allocattr(fgp);
                mutex_enter(&cp->c_statelock);
                if (error) {
                        error = 0;
                        goto out;
                }
        }

        if (cp->c_flags & CN_ALLOC_PENDING) {
                error = filegrp_create_metadata(fgp, &cp->c_metadata,
                    &cp->c_id);
                if (error)
                        goto out;
                cp->c_flags &= ~CN_ALLOC_PENDING;
        }

        if (cp->c_flags & CN_NEED_FRONT_SYNC) {
                if (cp->c_frontvp != NULL) {
                        error = VOP_FSYNC(cp->c_frontvp, FSYNC, kcred, NULL);
                        if (error) {
                                cp->c_metadata.md_timestamp.tv_sec = 0;
                        } else {
                                va.va_mask = AT_MTIME;
                                error = VOP_GETATTR(cp->c_frontvp, &va, 0,
                                    kcred, NULL);
                                if (error)
                                        goto out;
                                cp->c_metadata.md_timestamp = va.va_mtime;
                                cp->c_flags &=
                                    ~(CN_NEED_FRONT_SYNC |
                                    CN_POPULATION_PENDING);
                        }
                } else {
                        cp->c_flags &=
                            ~(CN_NEED_FRONT_SYNC | CN_POPULATION_PENDING);
                }
        }

        /*
         * XXX tony: How can CN_ALLOC_PENDING still be set??
         * XXX tony: How can CN_UPDATED not be set?????
```



```
         */
        if ((cp->c_flags & CN_ALLOC_PENDING) == 0 &&
            (cp->c_flags & CN_UPDATED)) {
                error = filegrp_write_metadata(fgp, &cp->c_id,
                    &cp->c_metadata);
                if (error)
                        goto out;
        }
out:
        if (error) {
                /* XXX modified files? */
                if (cp->c_metadata.md_rlno) {
                        cachefs_removefrontfile(&cp->c_metadata,
                            &cp->c_id, fgp);
                        cachefs_rlent_moveto(C_TO_FSCACHE(cp)->fs_cache,
                            CACHEFS_RL_FREE, cp->c_metadata.md_rlno, 0);
                        cp->c_metadata.md_rlno = 0;
                        cp->c_metadata.md_rltype = CACHEFS_RL_NONE;
                        if (cp->c_frontvp) {
                                VN_RELE(cp->c_frontvp);
                                cp->c_frontvp = NULL;
                        }
                }
                if ((cp->c_flags & CN_ALLOC_PENDING) == 0)
                        (void) filegrp_destroy_metadata(fgp, &cp->c_id);
                cp->c_flags |= CN_ALLOC_PENDING;
                cachefs_nocache(cp);
        }
        /*
         * we clear the updated bit even on errors because a retry
         * will probably fail also.
         */
        cp->c_flags &= ~CN_UPDATED;
        mutex_exit(&cp->c_statelock);

#ifdef CFSDEBUG
        CFS_DEBUG(CFSDEBUG_VOPS)
                printf("c_sync_metadata: EXIT cp %p cflag %x\n",
                    (void *)cp, cp->c_flags);
#endif

        return (error);
}

/*
 * This is the vop entry point for inactivating a vnode.
 * It just queues the request for the async thread which
 * calls cachefs_inactive.
 * Because of the dnlc, it is not safe to grab most locks here.
 */
/*ARGSUSED*/
static void
cachefs_inactive(struct vnode *vp, cred_t *cr, caller_context_t *ct)
{
        cnode_t *cp;
        struct cachefs_req *rp;
        fscache_t *fscp;

#ifdef CFSDEBUG
        CFS_DEBUG(CFSDEBUG_VOPS)
                printf("cachefs_inactive: ENTER vp %p\n", (void *)vp);
#endif

        cp = VTOC(vp);
        fscp = C_TO_FSCACHE(cp);
```

```
3428         ASSERT((cp->c_flags & CN_IDLE) == 0);

3430         /*
3431          * Cachefs only provides pass-through support for NFSv4,
3432          * and all vnode operations are passed through to the
3433          * back file system. For NFSv4 pass-through to work, only
3434          * connected operation is supported, the cnode backvp must
3435          * exist, and cachefs optional (eg., disconnectable) flags
3436          * are turned off. Assert these conditions to ensure that
3437          * the backfilesystem is called for the inactive operation.
3438          */
3439         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
3440         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

3442         /* vn_rele() set the v_count == 1 */

3444         cp->c_ipending = 1;

3446         rp = kmem_cache_alloc(cachefs_req_cache, KM_SLEEP);
3447         rp->cfs_cmd = CFS_IDLE;
3448         rp->cfs_cr = cr;
3449         crhold(rp->cfs_cr);
3450         rp->cfs_req_u.cu_idle.ci_vp = vp;
3451         cachefs_addqueue(rp, &(C_TO_FSCACHE(cp)->fs_workq));

3453 #ifdef CFSDEBUG
3454         CFS_DEBUG(CFSDEBUG_VOPS)
3455                 printf("cachefs_inactive: EXIT vp %p\n", (void *)vp);
3456 #endif
3457 }

3459 /* ARGSUSED */
3460 static int
3461 cachefs_lookup(vnode_t *dvp, char *nm, vnode_t **vpp,
3462     struct pathname *pnp, int flags, vnode_t *rdir, cred_t *cr,
3463     caller_context_t *ct, int *direntflags, pathname_t *realpnp)

3465 {
3466         int error = 0;
3467         cnode_t *dcp = VTOC(dvp);
3468         fscache_t *fscp = C_TO_FSCACHE(dcp);
3469         int held = 0;
3470         int connected = 0;

3472 #ifdef CFSDEBUG
3473         CFS_DEBUG(CFSDEBUG_VOPS)
3474                 printf("cachefs_lookup: ENTER dvp %p nm %s\n", (void *)dvp, nm);
3475 #endif

3477         if (getzoneid() != GLOBAL_ZONEID) {
3478                 error = EPERM;
3479                 goto out;
3480         }

3482         /*
3483          * Cachefs only provides pass-through support for NFSv4,
3484          * and all vnode operations are passed through to the
3485          * back file system. For NFSv4 pass-through to work, only
3486          * connected operation is supported, the cnode backvp must
3487          * exist, and cachefs optional (eg., disconnectable) flags
3488          * are turned off. Assert these conditions to ensure that
3489          * the backfilesystem is called for the lookup operation.
3490          */
3491         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
3492         CFS_BACKFS_NFSV4_ASSERT_CNODE(dcp);
```



```
3494         for (;;) {
3495                 /* get (or renew) access to the file system */
3496                 if (held) {
3497                         /* Won't loop with NFSv4 connected behavior */
3498                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
3499                         cachefs_cd_release(fscp);
3500                         held = 0;
3501                 }
3502                 error = cachefs_cd_access(fscp, connected, 0);
3503                 if (error)
3504                         break;
3505                 held = 1;

3507                 error = cachefs_lookup_common(dvp, nm, vpp, pnp,
3508                         flags, rdir, cr);
3509                 if (CFS_TIMEOUT(fscp, error)) {
3510                         if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
3511                                 cachefs_cd_release(fscp);
3512                                 held = 0;
3513                                 cachefs_cd_timedout(fscp);
3514                                 connected = 0;
3515                                 continue;
3516                         } else {
3517                                 if (cachefs_cd_access_miss(fscp)) {
3518                                         rw_enter(&dcp->c_rwlock, RW_READER);
3519                                         error = cachefs_lookup_back(dvp, nm,
3520                                             vpp, cr);
3521                                         rw_exit(&dcp->c_rwlock);
3522                                         if (!CFS_TIMEOUT(fscp, error))
3523                                                 break;
3524                                         delay(5*hz);
3525                                         connected = 0;
3526                                         continue;
3527                                 }
3528                                 connected = 1;
3529                                 continue;
3530                         }
3531                 }
3532                 break;
3533         }
3534         if (held)
3535                 cachefs_cd_release(fscp);

3537         if (error == 0 && IS_DEVVP(*vpp)) {
3538                 struct vnode *newvp;
3539                 newvp = specvp(*vpp, (*vpp)->v_rdev, (*vpp)->v_type, cr);
3540                 VN_RELE(*vpp);
3541                 if (newvp == NULL) {
3542                         error = ENOSYS;
3543                 } else {
3544                         *vpp = newvp;
3545                 }
3546         }

3548 #ifdef CFS_CD_DEBUG
3549         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
3550 #endif
3551 out:
3552 #ifdef CFSDEBUG
3553         CFS_DEBUG(CFSDEBUG_VOPS)
3554                 printf("cachefs_lookup: EXIT error = %d\n", error);
3555 #endif

3557         return (error);
3558 }
```

```
3560 /* ARGSUSED */
3561 int
3562 cachefs_lookup_common(vnode_t *dvp, char *nm, vnode_t **vpp,
3563     struct pathname *pnp, int flags, vnode_t *rdir, cred_t *cr)
3564 {
3565         int error = 0;
3566         cnode_t *cp, *dcp = VTOC(dvp);
3567         fscache_t *fscp = C_TO_FSCACHE(dcp);
3568         struct fid cookie;
3569         u_offset_t d_offset;
3570         struct cachefs_req *rp;
3571         cfs_cid_t cid, dircid;
3572         uint_t flag;
3573         uint_t uncached = 0;

3575         *vpp = NULL;

3577         /*
3578          * If lookup is for "", just return dvp.  Don't need
3579          * to send it over the wire, look it up in the dnlc,
3580          * or perform any access checks.
3581          */
3582         if (*nm == '\0') {
3583                 VN_HOLD(dvp);
3584                 *vpp = dvp;
3585                 return (0);
3586         }

3588         /* can't do lookups in non-directories */
3589         if (dvp->v_type != VDIR)
3590                 return (ENOTDIR);

3592         /* perform access check, also does consistency check if connected */
3593         if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
3594                 error = cachefs_access_connected(dvp, VEXEC, 0, cr);
3595         } else {
3596                 mutex_enter(&dcp->c_statelock);
3597                 error = cachefs_access_local(dcp, VEXEC, cr);
3598                 mutex_exit(&dcp->c_statelock);
3599         }
3600         if (error)
3601                 return (error);

3603         /*
3604          * If lookup is for ".", just return dvp.  Don't need
3605          * to send it over the wire or look it up in the dnlc,
3606          * just need to check access.
3607          */
3608         if (strcmp(nm, ".") == 0) {
3609                 VN_HOLD(dvp);
3610                 *vpp = dvp;
3611                 return (0);
3612         }

3614         /* check the dnlc */
3615         *vpp = (vnode_t *)dnlc_lookup(dvp, nm);
3616         if (*vpp)
3617                 return (0);

3619         /* read lock the dir before starting the search */
3620         rw_enter(&dcp->c_rwlock, RW_READER);

3622         mutex_enter(&dcp->c_statelock);
3623         dircid = dcp->c_id;

3625         dcp->c_usage++;
```



```
3627         /* if front file is not usable, lookup on the back fs */
3628         if ((dcp->c_flags & (CN_NOCACHE | CN_ASYNC_POPULATE)) ||
3629             CFS_ISFS_BACKFS_NFSV4(fscp) ||
3630             ((dcp->c_filegrp->fg_flags & CFS_FG_READ) == 0)) {
3631                 mutex_exit(&dcp->c_statelock);
3632                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED)
3633                         error = cachefs_lookup_back(dvp, nm, vpp, cr);
3634                 else
3635                         error = ETIMEDOUT;
3636                 goto out;
3637         }

3639         /* if the front file is not populated, try to populate it */
3640         if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
3641                 if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
3642                         error = ETIMEDOUT;
3643                         mutex_exit(&dcp->c_statelock);
3644                         goto out;
3645                 }

3647                 if (cachefs_async_okay()) {
3648                         /* cannot populate if cache is not writable */
3649                         ASSERT((dcp->c_flags &
3650                             (CN_ASYNC_POPULATE | CN_NOCACHE)) == 0);
3651                         dcp->c_flags |= CN_ASYNC_POPULATE;

3653                         rp = kmem_cache_alloc(cachefs_req_cache, KM_SLEEP);
3654                         rp->cfs_cmd = CFS_POPULATE;
3655                         rp->cfs_req_u.cu_populate.cpop_vp = dvp;
3656                         rp->cfs_cr = cr;

3658                         crhold(cr);
3659                         VN_HOLD(dvp);

3661                         cachefs_addqueue(rp, &fscp->fs_workq);
3662                 } else if (fscp->fs_info.fi_mntflags & CFS_NOACL) {
3663                         error = cachefs_dir_fill(dcp, cr);
3664                         if (error != 0) {
3665                                 mutex_exit(&dcp->c_statelock);
3666                                 goto out;
3667                         }
3668                 }
3669                 /* no populate if too many asyncs and we have to cache ACLs */

3671                 mutex_exit(&dcp->c_statelock);

3673                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED)
3674                         error = cachefs_lookup_back(dvp, nm, vpp, cr);
3675                 else
3676                         error = ETIMEDOUT;
3677                 goto out;
3678         }

3680         /* by now we have a valid cached front file that we can search */

3682         ASSERT((dcp->c_flags & CN_ASYNC_POPULATE) == 0);
3683         error = cachefs_dir_look(dcp, nm, &cookie, &flag,
3684             &d_offset, &cid);
3685         mutex_exit(&dcp->c_statelock);

3687         if (error) {
3688                 /* if the entry does not have the fid, go get it */
3689                 if (error == EINVAL) {
3690                         if (fscp->fs_cdconnected == CFS_CD_CONNECTED)
3691                                 error = cachefs_lookup_back(dvp, nm, vpp, cr);
```

```
3692                        else
3693                                error = ETIMEDOUT;
3694                }

3696                /* errors other than does not exist */
3697                else if (error != ENOENT) {
3698                        if (fscp->fs_cdconnected == CFS_CD_CONNECTED)
3699                                error = cachefs_lookup_back(dvp, nm, vpp, cr);
3700                        else
3701                                error = ETIMEDOUT;
3702                }
3703                goto out;
3704        }

3706        /*
3707         * Else we found the entry in the cached directory.
3708         * Make a cnode for it.
3709         */
3710        error = cachefs_cnode_make(&cid, fscp, &cookie, NULL, NULL,
3711            cr, 0, &cp);
3712        if (error == ESTALE) {
3713                ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
3714                mutex_enter(&dcp->c_statelock);
3715                cachefs_nocache(dcp);
3716                mutex_exit(&dcp->c_statelock);
3717                if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
3718                        error = cachefs_lookup_back(dvp, nm, vpp, cr);
3719                        uncached = 1;
3720                } else
3721                        error = ETIMEDOUT;
3722        } else if (error == 0) {
3723                *vpp = CTOV(cp);
3724        }

3726 out:
3727        if (error == 0) {
3728                /* put the entry in the dnlc */
3729                if (cachefs_dnlc)
3730                        dnlc_enter(dvp, nm, *vpp);

3732                /* save the cid of the parent so can find the name */
3733                cp = VTOC(*vpp);
3734                if (bcmp(&cp->c_metadata.md_parent, &dircid,
3735                    sizeof (cfs_cid_t)) != 0) {
3736                        mutex_enter(&cp->c_statelock);
3737                        cp->c_metadata.md_parent = dircid;
3738                        cp->c_flags |= CN_UPDATED;
3739                        mutex_exit(&cp->c_statelock);
3740                }
3741        }

3743        rw_exit(&dcp->c_rwlock);
3744        if (uncached && dcp->c_metadata.md_flags & MD_PACKED)
3745                (void) cachefs_pack_common(dvp, cr);
3746        return (error);
3747 }

3749 /*
3750  * Called from cachefs_lookup_common when the back file system needs to be
3751  * examined to perform the lookup.
3752  */
3753 static int
3754 cachefs_lookup_back(vnode_t *dvp, char *nm, vnode_t **vpp,
3755     cred_t *cr)
3756 {
3757        int error = 0;
```



```
3758        cnode_t *cp, *dcp = VTOC(dvp);
3759        fscache_t *fscp = C_TO_FSCACHE(dcp);
3760        vnode_t *backvp = NULL;
3761        struct vattr va;
3762        struct fid cookie;
3763        cfs_cid_t cid;
3764        uint32_t valid_fid;

3766        mutex_enter(&dcp->c_statelock);

3768        /* do a lookup on the back FS to get the back vnode */
3769        if (dcp->c_backvp == NULL) {
3770                error = cachefs_getbackvp(fscp, dcp);
3771                if (error)
3772                        goto out;
3773        }

3775        CFS_DPRINT_BACKFS_NFSV4(fscp,
3776            ("cachefs_lookup (nfsv4): dcp %p, dbackvp %p, name %s\n",
3777            dcp, dcp->c_backvp, nm));
3778        error = VOP_LOOKUP(dcp->c_backvp, nm, &backvp, (struct pathname *)NULL,
3779            0, (vnode_t *)NULL, cr, NULL, NULL, NULL);
3780        if (error)
3781                goto out;
3782        if (IS_DEVVP(backvp)) {
3783                struct vnode *devvp = backvp;

3785                if (VOP_REALVP(devvp, &backvp, NULL) == 0) {
3786                        VN_HOLD(backvp);
3787                        VN_RELE(devvp);
3788                }
3789        }

3791        /* get the fid and attrs from the back fs */
3792        valid_fid = (CFS_ISFS_BACKFS_NFSV4(fscp) ? FALSE : TRUE);
3793        error = cachefs_getcookie(backvp, &cookie, &va, cr, valid_fid);
3794        if (error)
3795                goto out;

3797        cid.cid_fileno = va.va_nodeid;
3798        cid.cid_flags = 0;

3800 #if 0
3801        /* XXX bob: this is probably no longer necessary */
3802        /* if the directory entry was incomplete, we can complete it now */
3803        if ((dcp->c_metadata.md_flags & MD_POPULATED) &&
3804            ((dcp->c_flags & CN_ASYNC_POPULATE) == 0) &&
3805            (dcp->c_filegrp->fg_flags & CFS_FG_WRITE)) {
3806                cachefs_dir_modentry(dcp, d_offset, &cookie, &cid);
3807        }
3808 #endif

3810 out:
3811        mutex_exit(&dcp->c_statelock);

3813        /* create the cnode */
3814        if (error == 0) {
3815                error = cachefs_cnode_make(&cid, fscp,
3816                    (valid_fid ? &cookie : NULL),
3817                    &va, backvp, cr, 0, &cp);
3818                if (error == 0) {
3819                        *vpp = CTOV(cp);
3820                }
3821        }

3823        if (backvp)
```

```
3824                 VN_RELE(backvp);

3826         return (error);
3827 }

3829 /*ARGSUSED7*/
3830 static int
3831 cachefs_create(vnode_t *dvp, char *nm, vattr_t *vap,
3832     vcexcl_t exclusive, int mode, vnode_t **vpp, cred_t *cr, int flag,
3833     caller_context_t *ct, vsecattr_t *vsecp)

3835 {
3836         cnode_t *dcp = VTOC(dvp);
3837         fscache_t *fscp = C_TO_FSCACHE(dcp);
3838         cachefscache_t *cachep = fscp->fs_cache;
3839         int error;
3840         int connected = 0;
3841         int held = 0;

3843 #ifdef CFSDEBUG
3844         CFS_DEBUG(CFSDEBUG_VOPS)
3845                 printf("cachefs_create: ENTER dvp %p, nm %s\n",
3846                     (void *)dvp, nm);
3847 #endif
3848         if (getzoneid() != GLOBAL_ZONEID) {
3849                 error = EPERM;
3850                 goto out;
3851         }

3853         /*
3854          * Cachefs only provides pass-through support for NFSv4,
3855          * and all vnode operations are passed through to the
3856          * back file system. For NFSv4 pass-through to work, only
3857          * connected operation is supported, the cnode backvp must
3858          * exist, and cachefs optional (eg., disconnectable) flags
3859          * are turned off. Assert these conditions to ensure that
3860          * the backfilesystem is called for the create operation.
3861          */
3862         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
3863         CFS_BACKFS_NFSV4_ASSERT_CNODE(dcp);

3865         for (;;) {
3866                 /* get (or renew) access to the file system */
3867                 if (held) {
3868                         /* Won't loop with NFSv4 connected behavior */
3869                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
3870                         cachefs_cd_release(fscp);
3871                         held = 0;
3872                 }
3873                 error = cachefs_cd_access(fscp, connected, 1);
3874                 if (error)
3875                         break;
3876                 held = 1;

3878                 /*
3879                  * if we are connected, perform the remote portion of the
3880                  * create.
3881                  */
3882                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
3883                         error = cachefs_create_connected(dvp, nm, vap,
3884                             exclusive, mode, vpp, cr);
3885                         if (CFS_TIMEOUT(fscp, error)) {
3886                                 cachefs_cd_release(fscp);
3887                                 held = 0;
3888                                 cachefs_cd_timedout(fscp);
3889                                 connected = 0;
```



```
3890                                 continue;
3891                         } else if (error) {
3892                                 break;
3893                         }
3894                 }

3896                 /* else we must be disconnected */
3897                 else {
3898                         error = cachefs_create_disconnected(dvp, nm, vap,
3899                             exclusive, mode, vpp, cr);
3900                         if (CFS_TIMEOUT(fscp, error)) {
3901                                 connected = 1;
3902                                 continue;
3903                         } else if (error) {
3904                                 break;
3905                         }
3906                 }
3907                 break;
3908         }

3910         if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_CREATE)) {
3911                 fid_t *fidp = NULL;
3912                 ino64_t fileno = 0;
3913                 cnode_t *cp = NULL;
3914                 if (error == 0)
3915                         cp = VTOC(*vpp);

3917                 if (cp != NULL) {
3918                         fidp = &cp->c_metadata.md_cookie;
3919                         fileno = cp->c_id.cid_fileno;
3920                 }
3921                 cachefs_log_create(cachep, error, fscp->fs_cfsvfsp,
3922                     fidp, fileno, crgetuid(cr));
3923         }

3925         if (held)
3926                 cachefs_cd_release(fscp);

3928         if (error == 0 && CFS_ISFS_NONSHARED(fscp))
3929                 (void) cachefs_pack(dvp, nm, cr);
3930         if (error == 0 && IS_DEVVP(*vpp)) {
3931                 struct vnode *spcvp;

3933                 spcvp = specvp(*vpp, (*vpp)->v_rdev, (*vpp)->v_type, cr);
3934                 VN_RELE(*vpp);
3935                 if (spcvp == NULL) {
3936                         error = ENOSYS;
3937                 } else {
3938                         *vpp = spcvp;
3939                 }
3940         }

3942 #ifdef CFS_CD_DEBUG
3943         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
3944 #endif
3945 out:
3946 #ifdef CFSDEBUG
3947         CFS_DEBUG(CFSDEBUG_VOPS)
3948                 printf("cachefs_create: EXIT error %d\n", error);
3949 #endif
3950         return (error);
3951 }


3954 static int
3955 cachefs_create_connected(vnode_t *dvp, char *nm, vattr_t *vap,
```

```
3956     enum vcexcl exclusive, int mode, vnode_t **vpp, cred_t *cr)
3957 {
3958         cnode_t *dcp = VTOC(dvp);
3959         fscache_t *fscp = C_TO_FSCACHE(dcp);
3960         int error;
3961         vnode_t *tvp = NULL;
3962         vnode_t *devvp;
3963         fid_t cookie;
3964         vattr_t va;
3965         cnode_t *ncp;
3966         cfs_cid_t cid;
3967         vnode_t *vp;
3968         uint32_t valid_fid;

3970         /* special case if file already exists */
3971         error = cachefs_lookup_common(dvp, nm, &vp, NULL, 0, NULL, cr);
3972         if (CFS_TIMEOUT(fscp, error))
3973                 return (error);
3974         if (error == 0) {
3975                 if (exclusive == EXCL)
3976                         error = EEXIST;
3977                 else if (vp->v_type == VDIR && (mode & VWRITE))
3978                         error = EISDIR;
3979                 else if ((error =
3980                     cachefs_access_connected(vp, mode, 0, cr)) == 0) {
3981                         if ((vap->va_mask & AT_SIZE) && (vp->v_type == VREG)) {
3982                                 vap->va_mask = AT_SIZE;
3983                                 error = cachefs_setattr_common(vp, vap, 0,
3984                                     cr, NULL);
3985                         }
3986                 }
3987                 if (error) {
3988                         VN_RELE(vp);
3989                 } else
3990                         *vpp = vp;
3991                 return (error);
3992         }

3994         rw_enter(&dcp->c_rwlock, RW_WRITER);
3995         mutex_enter(&dcp->c_statelock);

3997         /* consistency check the directory */
3998         error = CFSOP_CHECK_COBJECT(fscp, dcp, 0, cr);
3999         if (error) {
4000                 mutex_exit(&dcp->c_statelock);
4001                 goto out;
4002         }

4004         /* get the backvp if necessary */
4005         if (dcp->c_backvp == NULL) {
4006                 error = cachefs_getbackvp(fscp, dcp);
4007                 if (error) {
4008                         mutex_exit(&dcp->c_statelock);
4009                         goto out;
4010                 }
4011         }

4013         /* create the file on the back fs */
4014         CFS_DPRINT_BACKFS_NFSV4(fscp,
4015             ("cachefs_create (nfsv4): dcp %p, dbackvp %p,"
4016             "name %s\n", dcp, dcp->c_backvp, nm));
4017         error = VOP_CREATE(dcp->c_backvp, nm, vap, exclusive, mode,
4018             &devvp, cr, 0, NULL, NULL);
4019         mutex_exit(&dcp->c_statelock);
4020         if (error)
4021                 goto out;
```



```
4022         if (VOP_REALVP(devvp, &tvp, NULL) == 0) {
4023                 VN_HOLD(tvp);
4024                 VN_RELE(devvp);
4025         } else {
4026                 tvp = devvp;
4027         }

4029         /* get the fid and attrs from the back fs */
4030         valid_fid = (CFS_ISFS_BACKFS_NFSV4(fscp) ? FALSE : TRUE);
4031         error = cachefs_getcookie(tvp, &cookie, &va, cr, valid_fid);
4032         if (error)
4033                 goto out;

4035         /* make the cnode */
4036         cid.cid_fileno = va.va_nodeid;
4037         cid.cid_flags = 0;
4038         error = cachefs_cnode_make(&cid, fscp, (valid_fid ? &cookie : NULL),
4039             &va, tvp, cr, 0, &ncp);
4040         if (error)
4041                 goto out;

4043         *vpp = CTOV(ncp);

4045         /* enter it in the parent directory */
4046         mutex_enter(&dcp->c_statelock);
4047         if (CFS_ISFS_NONSHARED(fscp) &&
4048             (dcp->c_metadata.md_flags & MD_POPULATED)) {
4049                 /* see if entry already exists */
4050                 ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
4051                 error = cachefs_dir_look(dcp, nm, NULL, NULL, NULL, NULL);
4052                 if (error == ENOENT) {
4053                         /* entry, does not exist, add the new file */
4054                         error = cachefs_dir_enter(dcp, nm, &ncp->c_cookie,
4055                             &ncp->c_id, SM_ASYNC);
4056                         if (error) {
4057                                 cachefs_nocache(dcp);
4058                                 error = 0;
4059                         }
4060                         /* XXX should this be done elsewhere, too? */
4061                         dnlc_enter(dvp, nm, *vpp);
4062                 } else {
4063                         /* entry exists or some other problem */
4064                         cachefs_nocache(dcp);
4065                         error = 0;
4066                 }
4067         }
4068         CFSOP_MODIFY_COBJECT(fscp, dcp, cr);
4069         mutex_exit(&dcp->c_statelock);

4071 out:
4072         rw_exit(&dcp->c_rwlock);
4073         if (tvp)
4074                 VN_RELE(tvp);

4076         return (error);
4077 }

4079 static int
4080 cachefs_create_disconnected(vnode_t *dvp, char *nm, vattr_t *vap,
4081         enum vcexcl exclusive, int mode, vnode_t **vpp, cred_t *cr)
4082 {
4083         cnode_t *dcp = VTOC(dvp);
4084         cnode_t *cp;
4085         cnode_t *ncp = NULL;
4086         vnode_t *vp;
4087         fscache_t *fscp = C_TO_FSCACHE(dcp);
```

```
4088        int error = 0;
4089        struct vattr va;
4090        timestruc_t current_time;
4091        off_t commit = 0;
4092        fid_t cookie;
4093        cfs_cid_t cid;

4095        rw_enter(&dcp->c_rwlock, RW_WRITER);
4096        mutex_enter(&dcp->c_statelock);

4098        /* give up if the directory is not populated */
4099        if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
4100                mutex_exit(&dcp->c_statelock);
4101                rw_exit(&dcp->c_rwlock);
4102                return (ETIMEDOUT);
4103        }

4105        /* special case if file already exists */
4106        error = cachefs_dir_look(dcp, nm, &cookie, NULL, NULL, &cid);
4107        if (error == EINVAL) {
4108                mutex_exit(&dcp->c_statelock);
4109                rw_exit(&dcp->c_rwlock);
4110                return (ETIMEDOUT);
4111        }
4112        if (error == 0) {
4113                mutex_exit(&dcp->c_statelock);
4114                rw_exit(&dcp->c_rwlock);
4115                error = cachefs_cnode_make(&cid, fscp, &cookie, NULL, NULL,
4116                    cr, 0, &cp);
4117                if (error) {
4118                        return (error);
4119                }
4120                vp = CTOV(cp);

4122                if (cp->c_metadata.md_flags & MD_NEEDATTRS)
4123                        error = ETIMEDOUT;
4124                else if (exclusive == EXCL)
4125                        error = EEXIST;
4126                else if (vp->v_type == VDIR && (mode & VWRITE))
4127                        error = EISDIR;
4128                else {
4129                        mutex_enter(&cp->c_statelock);
4130                        error = cachefs_access_local(cp, mode, cr);
4131                        mutex_exit(&cp->c_statelock);
4132                        if (!error) {
4133                                if ((vap->va_mask & AT_SIZE) &&
4134                                    (vp->v_type == VREG)) {
4135                                        vap->va_mask = AT_SIZE;
4136                                        error = cachefs_setattr_common(vp,
4137                                            vap, 0, cr, NULL);
4138                                }
4139                        }
4140                }
4141                if (error) {
4142                        VN_RELE(vp);
4143                } else
4144                        *vpp = vp;
4145                return (error);
4146        }

4148        /* give up if cannot modify the cache */
4149        if (CFS_ISFS_WRITE_AROUND(fscp)) {
4150                mutex_exit(&dcp->c_statelock);
4151                error = ETIMEDOUT;
4152                goto out;
4153        }
```



```
4155        /* check access */
4156        if (error = cachefs_access_local(dcp, VWRITE, cr)) {
4157                mutex_exit(&dcp->c_statelock);
4158                goto out;
4159        }

4161        /* mark dir as modified */
4162        cachefs_modified(dcp);
4163        mutex_exit(&dcp->c_statelock);

4165        /* must be privileged to set sticky bit */
4166        if ((vap->va_mode & VSVTX) && secpolicy_vnode_stky_modify(cr) != 0)
4167                vap->va_mode &= ~VSVTX;

4169        /* make up a reasonable set of attributes */
4170        cachefs_attr_setup(vap, &va, dcp, cr);

4172        /* create the cnode */
4173        error = cachefs_cnode_create(fscp, &va, 0, &ncp);
4174        if (error)
4175                goto out;

4177        mutex_enter(&ncp->c_statelock);

4179        /* get the front file now instead of later */
4180        if (vap->va_type == VREG) {
4181                error = cachefs_getfrontfile(ncp);
4182                if (error) {
4183                        mutex_exit(&ncp->c_statelock);
4184                        goto out;
4185                }
4186                ASSERT(ncp->c_frontvp != NULL);
4187                ASSERT((ncp->c_flags & CN_ALLOC_PENDING) == 0);
4188                ncp->c_metadata.md_flags |= MD_POPULATED;
4189        } else {
4190                ASSERT(ncp->c_flags & CN_ALLOC_PENDING);
4191                if (ncp->c_filegrp->fg_flags & CFS_FG_ALLOC_ATTR) {
4192                        (void) filegrp_allocattr(ncp->c_filegrp);
4193                }
4194                error = filegrp_create_metadata(ncp->c_filegrp,
4195                    &ncp->c_metadata, &ncp->c_id);
4196                if (error) {
4197                        mutex_exit(&ncp->c_statelock);
4198                        goto out;
4199                }
4200                ncp->c_flags &= ~CN_ALLOC_PENDING;
4201        }
4202        mutex_enter(&dcp->c_statelock);
4203        cachefs_creategid(dcp, ncp, vap, cr);
4204        cachefs_createacl(dcp, ncp);
4205        mutex_exit(&dcp->c_statelock);

4207        /* set times on the file */
4208        gethrestime(&current_time);
4209        ncp->c_metadata.md_vattr.va_atime = current_time;
4210        ncp->c_metadata.md_localctime = current_time;
4211        ncp->c_metadata.md_localmtime = current_time;
4212        ncp->c_metadata.md_flags |= MD_LOCALMTIME | MD_LOCALCTIME;

4214        /* reserve space for the daemon cid mapping */
4215        error = cachefs_dlog_cidmap(fscp);
4216        if (error) {
4217                mutex_exit(&ncp->c_statelock);
4218                goto out;
4219        }
```

```
4220         ncp->c_metadata.md_flags |= MD_MAPPING;

4222         /* mark the new file as modified */
4223         if (cachefs_modified_alloc(ncp)) {
4224                 mutex_exit(&ncp->c_statelock);
4225                 error = ENOSPC;
4226                 goto out;
4227         }
4228         ncp->c_flags |= CN_UPDATED;

4230         /*
4231          * write the metadata now rather than waiting until
4232          * inactive so that if there's no space we can let
4233          * the caller know.
4234          */
4235         ASSERT((ncp->c_flags & CN_ALLOC_PENDING) == 0);
4236         ASSERT((ncp->c_filegrp->fg_flags & CFS_FG_ALLOC_ATTR) == 0);
4237         error = filegrp_write_metadata(ncp->c_filegrp,
4238             &ncp->c_id, &ncp->c_metadata);
4239         if (error) {
4240                 mutex_exit(&ncp->c_statelock);
4241                 goto out;
4242         }

4244         /* log the operation */
4245         commit = cachefs_dlog_create(fscp, dcp, nm, vap, exclusive,
4246             mode, ncp, 0, cr);
4247         if (commit == 0) {
4248                 mutex_exit(&ncp->c_statelock);
4249                 error = ENOSPC;
4250                 goto out;
4251         }

4253         mutex_exit(&ncp->c_statelock);

4255         mutex_enter(&dcp->c_statelock);

4257         /* update parent dir times */
4258         dcp->c_metadata.md_localmtime = current_time;
4259         dcp->c_metadata.md_flags |= MD_LOCALMTIME;
4260         dcp->c_flags |= CN_UPDATED;

4262         /* enter new file name in the parent directory */
4263         if (dcp->c_metadata.md_flags & MD_POPULATED) {
4264                 error = cachefs_dir_enter(dcp, nm, &ncp->c_cookie,
4265                     &ncp->c_id, 0);
4266                 if (error) {
4267                         cachefs_nocache(dcp);
4268                         mutex_exit(&dcp->c_statelock);
4269                         error = ETIMEDOUT;
4270                         goto out;
4271                 }
4272                 dnlc_enter(dvp, nm, CTOV(ncp));
4273         } else {
4274                 mutex_exit(&dcp->c_statelock);
4275                 error = ETIMEDOUT;
4276                 goto out;
4277         }
4278         mutex_exit(&dcp->c_statelock);

4280 out:
4281         rw_exit(&dcp->c_rwlock);

4283         if (commit) {
4284                 if (cachefs_dlog_commit(fscp, commit, error)) {
4285                         /*EMPTY*/
```



```
4286                         /* XXX bob: fix on panic */
4287                 }
4288         }
4289         if (error) {
4290                 /* destroy the cnode we created */
4291                 if (ncp) {
4292                         mutex_enter(&ncp->c_statelock);
4293                         ncp->c_flags |= CN_DESTROY;
4294                         mutex_exit(&ncp->c_statelock);
4295                         VN_RELE(CTOV(ncp));
4296                 }
4297         } else {
4298                 *vpp = CTOV(ncp);
4299         }
4300         return (error);
4301 }

4303 /*ARGSUSED*/
4304 static int
4305 cachefs_remove(vnode_t *dvp, char *nm, cred_t *cr, caller_context_t *ct,
4306     int flags)
4307 {
4308         cnode_t *dcp = VTOC(dvp);
4309         fscache_t *fscp = C_TO_FSCACHE(dcp);
4310         cachefscache_t *cachep = fscp->fs_cache;
4311         int error = 0;
4312         int held = 0;
4313         int connected = 0;
4314         size_t namlen;
4315         vnode_t *vp = NULL;
4316         int vfslock = 0;

4318 #ifdef CFSDEBUG
4319         CFS_DEBUG(CFSDEBUG_VOPS)
4320                 printf("cachefs_remove: ENTER dvp %p name %s\n",
4321                     (void *)dvp, nm);
4322 #endif
4323         if (getzoneid() != GLOBAL_ZONEID) {
4324                 error = EPERM;
4325                 goto out;
4326         }

4328         if (fscp->fs_cache->c_flags & (CACHE_NOFILL | CACHE_NOCACHE))
4329                 ASSERT(dcp->c_flags & CN_NOCACHE);

4331         /*
4332          * Cachefs only provides pass-through support for NFSv4,
4333          * and all vnode operations are passed through to the
4334          * back file system. For NFSv4 pass-through to work, only
4335          * connected operation is supported, the cnode backvp must
4336          * exist, and cachefs optional (eg., disconnectable) flags
4337          * are turned off. Assert these conditions to ensure that
4338          * the backfilesystem is called for the remove operation.
4339          */
4340         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
4341         CFS_BACKFS_NFSV4_ASSERT_CNODE(dcp);

4343         for (;;) {
4344                 if (vfslock) {
4345                         vn_vfsunlock(vp);
4346                         vfslock = 0;
4347                 }
4348                 if (vp) {
4349                         VN_RELE(vp);
4350                         vp = NULL;
4351                 }
```

```
4353                 /* get (or renew) access to the file system */
4354                 if (held) {
4355                         /* Won't loop with NFSv4 connected behavior */
4356                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
4357                         cachefs_cd_release(fscp);
4358                         held = 0;
4359                 }
4360                 error = cachefs_cd_access(fscp, connected, 1);
4361                 if (error)
4362                         break;
4363                 held = 1;

4365                 /* if disconnected, do some extra error checking */
4366                 if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
4367                         /* check permissions */
4368                         mutex_enter(&dcp->c_statelock);
4369                         error = cachefs_access_local(dcp, (VEXEC|VWRITE), cr);
4370                         mutex_exit(&dcp->c_statelock);
4371                         if (CFS_TIMEOUT(fscp, error)) {
4372                                 connected = 1;
4373                                 continue;
4374                         }
4375                         if (error)
4376                                 break;

4378                         namlen = strlen(nm);
4379                         if (namlen == 0) {
4380                                 error = EINVAL;
4381                                 break;
4382                         }

4384                         /* cannot remove . and .. */
4385                         if (nm[0] == '.') {
4386                                 if (namlen == 1) {
4387                                         error = EINVAL;
4388                                         break;
4389                                 } else if (namlen == 2 && nm[1] == '.') {
4390                                         error = EEXIST;
4391                                         break;
4392                                 }
4393                         }

4395                 }

4397                 /* get the cnode of the file to delete */
4398                 error = cachefs_lookup_common(dvp, nm, &vp, NULL, 0, NULL, cr);
4399                 if (error) {
4400                         if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
4401                                 if (CFS_TIMEOUT(fscp, error)) {
4402                                         cachefs_cd_release(fscp);
4403                                         held = 0;
4404                                         cachefs_cd_timedout(fscp);
4405                                         connected = 0;
4406                                         continue;
4407                                 }
4408                         } else {
4409                                 if (CFS_TIMEOUT(fscp, error)) {
4410                                         connected = 1;
4411                                         continue;
4412                                 }
4413                         }
4414                         if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_REMOVE)) {
4415                                 struct fid foo;

4417                                 bzero(&foo, sizeof (foo));
```



```
4418                                 cachefs_log_remove(cachep, error,
4419                                     fscp->fs_cfsvfsp, &foo, 0, crgetuid(cr));
4420                         }
4421                         break;
4422                 }

4424                 if (vp->v_type == VDIR) {
4425                         /* must be privileged to remove dirs with unlink() */
4426                         if ((error = secpolicy_fs_linkdir(cr, vp->v_vfsp)) != 0)
4427                                 break;

4429                         /* see ufs_dirremove for why this is done, mount race */
4430                         if (vn_vfswlock(vp)) {
4431                                 error = EBUSY;
4432                                 break;
4433                         }
4434                         vfslock = 1;
4435                         if (vn_mountedvfs(vp) != NULL) {
4436                                 error = EBUSY;
4437                                 break;
4438                         }
4439                 }

4441                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
4442                         error = cachefs_remove_connected(dvp, nm, cr, vp);
4443                         if (CFS_TIMEOUT(fscp, error)) {
4444                                 cachefs_cd_release(fscp);
4445                                 held = 0;
4446                                 cachefs_cd_timedout(fscp);
4447                                 connected = 0;
4448                                 continue;
4449                         }
4450                 } else {
4451                         error = cachefs_remove_disconnected(dvp, nm, cr,
4452                             vp);
4453                         if (CFS_TIMEOUT(fscp, error)) {
4454                                 connected = 1;
4455                                 continue;
4456                         }
4457                 }
4458                 break;
4459         }

4461 #if 0
4462         if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_REMOVE))
4463                 cachefs_log_remove(cachep, error, fscp->fs_cfsvfsp,
4464                     &cp->c_metadata.md_cookie, cp->c_id.cid_fileno,
4465                     crgetuid(cr));
4466 #endif

4468         if (held)
4469                 cachefs_cd_release(fscp);

4471         if (vfslock)
4472                 vn_vfsunlock(vp);

4474         if (vp)
4475                 VN_RELE(vp);

4477 #ifdef CFS_CD_DEBUG
4478         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
4479 #endif
4480 out:
4481 #ifdef CFSDEBUG
4482         CFS_DEBUG(CFSDEBUG_VOPS)
4483                 printf("cachefs_remove: EXIT dvp %p\n", (void *)dvp);
```

```
#endif

        return (error);
}

int
cachefs_remove_connected(vnode_t *dvp, char *nm, cred_t *cr, vnode_t *vp)
{
        cnode_t *dcp = VTOC(dvp);
        cnode_t *cp = VTOC(vp);
        fscache_t *fscp = C_TO_FSCACHE(dcp);
        int error = 0;

        /*
         * Acquire the rwlock (WRITER) on the directory to prevent other
         * activity on the directory.
         */
        rw_enter(&dcp->c_rwlock, RW_WRITER);

        /* purge dnlc of this entry so can get accurate vnode count */
        dnlc_purge_vp(vp);

        /*
         * If the cnode is active, make a link to the file
         * so operations on the file will continue.
         */
        if ((vp->v_type != VDIR) &&
            !((vp->v_count == 1) || ((vp->v_count == 2) && cp->c_ipending))) {
                error = cachefs_remove_dolink(dvp, vp, nm, cr);
                if (error)
                        goto out;
        }

        /* else call backfs NFSv4 handler if NFSv4 */
        else if (CFS_ISFS_BACKFS_NFSV4(fscp)) {
                error = cachefs_remove_backfs_nfsv4(dvp, nm, cr, vp);
                goto out;
        }

        /* else drop the backvp so nfs does not do rename */
        else if (cp->c_backvp) {
                mutex_enter(&cp->c_statelock);
                if (cp->c_backvp) {
                        VN_RELE(cp->c_backvp);
                        cp->c_backvp = NULL;
                }
                mutex_exit(&cp->c_statelock);
        }

        mutex_enter(&dcp->c_statelock);

        /* get the backvp */
        if (dcp->c_backvp == NULL) {
                error = cachefs_getbackvp(fscp, dcp);
                if (error) {
                        mutex_exit(&dcp->c_statelock);
                        goto out;
                }
        }

        /* check directory consistency */
        error = CFSOP_CHECK_COBJECT(fscp, dcp, 0, cr);
        if (error) {
                mutex_exit(&dcp->c_statelock);
                goto out;
        }
```



```
        /* perform the remove on the back fs */
        error = VOP_REMOVE(dcp->c_backvp, nm, cr, NULL, 0);
        if (error) {
                mutex_exit(&dcp->c_statelock);
                goto out;
        }

        /* the dir has been modified */
        CFSOP_MODIFY_COBJECT(fscp, dcp, cr);

        /* remove the entry from the populated directory */
        if (CFS_ISFS_NONSHARED(fscp) &&
            (dcp->c_metadata.md_flags & MD_POPULATED)) {
                error = cachefs_dir_rmentry(dcp, nm);
                if (error) {
                        cachefs_nocache(dcp);
                        error = 0;
                }
        }
        mutex_exit(&dcp->c_statelock);

        /* fix up the file we deleted */
        mutex_enter(&cp->c_statelock);
        if (cp->c_attr.va_nlink == 1)
                cp->c_flags |= CN_DESTROY;
        else
                cp->c_flags |= CN_UPDATED;

        cp->c_attr.va_nlink--;
        CFSOP_MODIFY_COBJECT(fscp, cp, cr);
        mutex_exit(&cp->c_statelock);

out:
        rw_exit(&dcp->c_rwlock);
        return (error);
}

/*
 * cachefs_remove_backfs_nfsv4
 *
 * Call NFSv4 back filesystem to handle the remove (cachefs
 * pass-through support for NFSv4).
 */
int
cachefs_remove_backfs_nfsv4(vnode_t *dvp, char *nm, cred_t *cr, vnode_t *vp)
{
        cnode_t *dcp = VTOC(dvp);
        cnode_t *cp = VTOC(vp);
        vnode_t *dbackvp;
        fscache_t *fscp = C_TO_FSCACHE(dcp);
        int error = 0;

        /*
         * For NFSv4 pass-through to work, only connected operation
         * is supported, the cnode backvp must exist, and cachefs
         * optional (eg., disconnectable) flags are turned off. Assert
         * these conditions for the getattr operation.
         */
        CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
        CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);

        /* Should hold the directory readwrite lock to update directory */
        ASSERT(RW_WRITE_HELD(&dcp->c_rwlock));

        /*
```

```
4616          * Update attributes for directory. Note that
4617          * CFSOP_CHECK_COBJECT asserts for c_statelock being
4618          * held, so grab it before calling the routine.
4619          */
4620         mutex_enter(&dcp->c_statelock);
4621         error = CFSOP_CHECK_COBJECT(fscp, dcp, 0, cr);
4622         mutex_exit(&dcp->c_statelock);
4623         if (error)
4624                 goto out;

4626         /*
4627          * Update attributes for cp. Note that CFSOP_CHECK_COBJECT
4628          * asserts for c_statelock being held, so grab it before
4629          * calling the routine.
4630          */
4631         mutex_enter(&cp->c_statelock);
4632         error = CFSOP_CHECK_COBJECT(fscp, cp, 0, cr);
4633         if (error) {
4634                 mutex_exit(&cp->c_statelock);
4635                 goto out;
4636         }

4638         /*
4639          * Drop the backvp so nfs if the link count is 1 so that
4640          * nfs does not do rename. Ensure that we will destroy the cnode
4641          * since this cnode no longer contains the backvp. Note that we
4642          * maintain lock on this cnode to prevent change till the remove
4643          * completes, otherwise other operations will encounter an ESTALE
4644          * if they try to use the cnode with CN_DESTROY set (see
4645          * cachefs_get_backvp()), or change the state of the cnode
4646          * while we're removing it.
4647          */
4648         if (cp->c_attr.va_nlink == 1) {
4649                 /*
4650                  * The unldvp information is created for the case
4651                  * when there is more than one reference on the
4652                  * vnode when a remove operation is called. If the
4653                  * remove itself was holding a reference to the
4654                  * vnode, then a subsequent remove will remove the
4655                  * backvp, so we need to get rid of the unldvp
4656                  * before removing the backvp. An alternate would
4657                  * be to simply ignore the remove and let the
4658                  * inactivation routine do the deletion of the
4659                  * unldvp.
4660                  */
4661                 if (cp->c_unldvp) {
4662                         VN_RELE(cp->c_unldvp);
4663                         cachefs_kmem_free(cp->c_unlname, MAXNAMELEN);
4664                         crfree(cp->c_unlcred);
4665                         cp->c_unldvp = NULL;
4666                         cp->c_unlcred = NULL;
4667                 }
4668                 cp->c_flags |= CN_DESTROY;
4669                 cp->c_attr.va_nlink = 0;
4670                 VN_RELE(cp->c_backvp);
4671                 cp->c_backvp = NULL;
4672         }

4674         /* perform the remove on back fs after extracting directory backvp */
4675         mutex_enter(&dcp->c_statelock);
4676         dbackvp = dcp->c_backvp;
4677         mutex_exit(&dcp->c_statelock);

4679         CFS_DPRINT_BACKFS_NFSV4(fscp,
4680             ("cachefs_remove (nfsv4): dcp %p, dbackvp %p, name %s\n",
4681                 dcp, dbackvp, nm));
```



```
4682         error = VOP_REMOVE(dbackvp, nm, cr, NULL, 0);
4683         if (error) {
4684                 mutex_exit(&cp->c_statelock);
4685                 goto out;
4686         }

4688         /* fix up the file we deleted, if not destroying the cnode */
4689         if ((cp->c_flags & CN_DESTROY) == 0) {
4690                 cp->c_attr.va_nlink--;
4691                 cp->c_flags |= CN_UPDATED;
4692         }

4694         mutex_exit(&cp->c_statelock);

4696 out:
4697         return (error);
4698 }

4700 int
4701 cachefs_remove_disconnected(vnode_t *dvp, char *nm, cred_t *cr,
4702     vnode_t *vp)
4703 {
4704         cnode_t *dcp = VTOC(dvp);
4705         cnode_t *cp = VTOC(vp);
4706         fscache_t *fscp = C_TO_FSCACHE(dcp);
4707         int error = 0;
4708         off_t commit = 0;
4709         timestruc_t current_time;

4711         if (CFS_ISFS_WRITE_AROUND(fscp))
4712                 return (ETIMEDOUT);

4714         if (cp->c_metadata.md_flags & MD_NEEDATTRS)
4715                 return (ETIMEDOUT);

4717         /*
4718          * Acquire the rwlock (WRITER) on the directory to prevent other
4719          * activity on the directory.
4720          */
4721         rw_enter(&dcp->c_rwlock, RW_WRITER);

4723         /* dir must be populated */
4724         if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
4725                 error = ETIMEDOUT;
4726                 goto out;
4727         }

4729         mutex_enter(&dcp->c_statelock);
4730         mutex_enter(&cp->c_statelock);

4732         error = cachefs_stickyrmchk(dcp, cp, cr);

4734         mutex_exit(&cp->c_statelock);
4735         mutex_exit(&dcp->c_statelock);
4736         if (error)
4737                 goto out;

4739         /* purge dnlc of this entry so can get accurate vnode count */
4740         dnlc_purge_vp(vp);

4742         /*
4743          * If the cnode is active, make a link to the file
4744          * so operations on the file will continue.
4745          */
4746         if ((vp->v_type != VDIR) &&
4747             !((vp->v_count == 1) || ((vp->v_count == 2) && cp->c_ipending))) {
```

```
4748                 error = cachefs_remove_dolink(dvp, vp, nm, cr);
4749                 if (error)
4750                         goto out;
4751         }

4753         if (cp->c_attr.va_nlink > 1) {
4754                 mutex_enter(&cp->c_statelock);
4755                 if (cachefs_modified_alloc(cp)) {
4756                         mutex_exit(&cp->c_statelock);
4757                         error = ENOSPC;
4758                         goto out;
4759                 }
4760                 if ((cp->c_metadata.md_flags & MD_MAPPING) == 0) {
4761                         error = cachefs_dlog_cidmap(fscp);
4762                         if (error) {
4763                                 mutex_exit(&cp->c_statelock);
4764                                 error = ENOSPC;
4765                                 goto out;
4766                         }
4767                         cp->c_metadata.md_flags |= MD_MAPPING;
4768                         cp->c_flags |= CN_UPDATED;
4769                 }
4770                 mutex_exit(&cp->c_statelock);
4771         }

4773         /* log the remove */
4774         commit = cachefs_dlog_remove(fscp, dcp, nm, cp, cr);
4775         if (commit == 0) {
4776                 error = ENOSPC;
4777                 goto out;
4778         }

4780         /* remove the file from the dir */
4781         mutex_enter(&dcp->c_statelock);
4782         if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
4783                 mutex_exit(&dcp->c_statelock);
4784                 error = ETIMEDOUT;
4785                 goto out;

4787         }
4788         cachefs_modified(dcp);
4789         error = cachefs_dir_rmentry(dcp, nm);
4790         if (error) {
4791                 mutex_exit(&dcp->c_statelock);
4792                 if (error == ENOTDIR)
4793                         error = ETIMEDOUT;
4794                 goto out;
4795         }

4797         /* update parent dir times */
4798         gethrestime(&current_time);
4799         dcp->c_metadata.md_localctime = current_time;
4800         dcp->c_metadata.md_localmtime = current_time;
4801         dcp->c_metadata.md_flags |= MD_LOCALCTIME | MD_LOCALMTIME;
4802         dcp->c_flags |= CN_UPDATED;
4803         mutex_exit(&dcp->c_statelock);

4805         /* adjust file we are deleting */
4806         mutex_enter(&cp->c_statelock);
4807         cp->c_attr.va_nlink--;
4808         cp->c_metadata.md_localctime = current_time;
4809         cp->c_metadata.md_flags |= MD_LOCALCTIME;
4810         if (cp->c_attr.va_nlink == 0) {
4811                 cp->c_flags |= CN_DESTROY;
4812         } else {
4813                 cp->c_flags |= CN_UPDATED;
```



```
4814         }
4815         mutex_exit(&cp->c_statelock);

4817 out:
4818         if (commit) {
4819                 /* commit the log entry */
4820                 if (cachefs_dlog_commit(fscp, commit, error)) {
4821                         /*EMPTY*/
4822                         /* XXX bob: fix on panic */
4823                 }
4824         }

4826         rw_exit(&dcp->c_rwlock);
4827         return (error);
4828 }

4830 /*ARGSUSED*/
4831 static int
4832 cachefs_link(vnode_t *tdvp, vnode_t *fvp, char *tnm, cred_t *cr,
4833     caller_context_t *ct, int flags)
4834 {
4835         fscache_t *fscp = VFS_TO_FSCACHE(tdvp->v_vfsp);
4836         cnode_t *tdcp = VTOC(tdvp);
4837         struct vnode *realvp;
4838         int error = 0;
4839         int held = 0;
4840         int connected = 0;

4842 #ifdef CFSDEBUG
4843         CFS_DEBUG(CFSDEBUG_VOPS)
4844                 printf("cachefs_link: ENTER fvp %p tdvp %p tnm %s\n",
4845                     (void *)fvp, (void *)tdvp, tnm);
4846 #endif

4848         if (getzoneid() != GLOBAL_ZONEID) {
4849                 error = EPERM;
4850                 goto out;
4851         }

4853         if (fscp->fs_cache->c_flags & (CACHE_NOFILL | CACHE_NOCACHE))
4854                 ASSERT(tdcp->c_flags & CN_NOCACHE);

4856         if (VOP_REALVP(fvp, &realvp, ct) == 0) {
4857                 fvp = realvp;
4858         }

4860         /*
4861          * Cachefs only provides pass-through support for NFSv4,
4862          * and all vnode operations are passed through to the
4863          * back file system. For NFSv4 pass-through to work, only
4864          * connected operation is supported, the cnode backvp must
4865          * exist, and cachefs optional (eg., disconnectable) flags
4866          * are turned off. Assert these conditions to ensure that
4867          * the backfilesystem is called for the link operation.
4868          */

4870         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
4871         CFS_BACKFS_NFSV4_ASSERT_CNODE(tdcp);

4873         for (;;) {
4874                 /* get (or renew) access to the file system */
4875                 if (held) {
4876                         /* Won't loop with NFSv4 connected behavior */
4877                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
4878                         rw_exit(&tdcp->c_rwlock);
4879                         cachefs_cd_release(fscp);
```

```
4880                        held = 0;
4881                }
4882                error = cachefs_cd_access(fscp, connected, 1);
4883                if (error)
4884                        break;
4885                rw_enter(&tdcp->c_rwlock, RW_WRITER);
4886                held = 1;

4888                if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
4889                        error = cachefs_link_connected(tdvp, fvp, tnm, cr);
4890                        if (CFS_TIMEOUT(fscp, error)) {
4891                                rw_exit(&tdcp->c_rwlock);
4892                                cachefs_cd_release(fscp);
4893                                held = 0;
4894                                cachefs_cd_timedout(fscp);
4895                                connected = 0;
4896                                continue;
4897                        }
4898                } else {
4899                        error = cachefs_link_disconnected(tdvp, fvp, tnm,
4900                            cr);
4901                        if (CFS_TIMEOUT(fscp, error)) {
4902                                connected = 1;
4903                                continue;
4904                        }
4905                }
4906                break;
4907        }

4909        if (held) {
4910                rw_exit(&tdcp->c_rwlock);
4911                cachefs_cd_release(fscp);
4912        }

4914 #ifdef CFS_CD_DEBUG
4915        ASSERT((curthread->t_flag & T_CD_HELD) == 0);
4916 #endif
4917 out:
4918 #ifdef CFSDEBUG
4919        CFS_DEBUG(CFSDEBUG_VOPS)
4920                printf("cachefs_link: EXIT fvp %p tdvp %p tnm %s\n",
4921                    (void *)fvp, (void *)tdvp, tnm);
4922 #endif
4923        return (error);
4924 }

4926 static int
4927 cachefs_link_connected(vnode_t *tdvp, vnode_t *fvp, char *tnm, cred_t *cr)
4928 {
4929        cnode_t *tdcp = VTOC(tdvp);
4930        cnode_t *fcp = VTOC(fvp);
4931        fscache_t *fscp = VFS_TO_FSCACHE(tdvp->v_vfsp);
4932        int error = 0;
4933        vnode_t *backvp = NULL;

4935        if (tdcp != fcp) {
4936                mutex_enter(&fcp->c_statelock);

4938                if (fcp->c_backvp == NULL) {
4939                        error = cachefs_getbackvp(fscp, fcp);
4940                        if (error) {
4941                                mutex_exit(&fcp->c_statelock);
4942                                goto out;
4943                        }
4944                }
```



```
4946                error = CFSOP_CHECK_COBJECT(fscp, fcp, 0, cr);
4947                if (error) {
4948                        mutex_exit(&fcp->c_statelock);
4949                        goto out;
4950                }
4951                backvp = fcp->c_backvp;
4952                VN_HOLD(backvp);
4953                mutex_exit(&fcp->c_statelock);
4954        }

4956        mutex_enter(&tdcp->c_statelock);

4958        /* get backvp of target directory */
4959        if (tdcp->c_backvp == NULL) {
4960                error = cachefs_getbackvp(fscp, tdcp);
4961                if (error) {
4962                        mutex_exit(&tdcp->c_statelock);
4963                        goto out;
4964                }
4965        }

4967        /* consistency check target directory */
4968        error = CFSOP_CHECK_COBJECT(fscp, tdcp, 0, cr);
4969        if (error) {
4970                mutex_exit(&tdcp->c_statelock);
4971                goto out;
4972        }
4973        if (backvp == NULL) {
4974                backvp = tdcp->c_backvp;
4975                VN_HOLD(backvp);
4976        }

4978        /* perform the link on the back fs */
4979        CFS_DPRINT_BACKFS_NFSV4(fscp,
4980            ("cachefs_link (nfsv4): tdcp %p, tdbackvp %p, "
4981            "name %s\n", tdcp, tdcp->c_backvp, tnm));
4982        error = VOP_LINK(tdcp->c_backvp, backvp, tnm, cr, NULL, 0);
4983        if (error) {
4984                mutex_exit(&tdcp->c_statelock);
4985                goto out;
4986        }

4988        CFSOP_MODIFY_COBJECT(fscp, tdcp, cr);

4990        /* if the dir is populated, add the new link */
4991        if (CFS_ISFS_NONSHARED(fscp) &&
4992            (tdcp->c_metadata.md_flags & MD_POPULATED)) {
4993                error = cachefs_dir_enter(tdcp, tnm, &fcp->c_cookie,
4994                    &fcp->c_id, SM_ASYNC);
4995                if (error) {
4996                        cachefs_nocache(tdcp);
4997                        error = 0;
4998                }
4999        }
5000        mutex_exit(&tdcp->c_statelock);

5002        /* get the new link count on the file */
5003        mutex_enter(&fcp->c_statelock);
5004        fcp->c_flags |= CN_UPDATED;
5005        CFSOP_MODIFY_COBJECT(fscp, fcp, cr);
5006        if (fcp->c_backvp == NULL) {
5007                error = cachefs_getbackvp(fscp, fcp);
5008                if (error) {
5009                        mutex_exit(&fcp->c_statelock);
5010                        goto out;
5011                }
```

```
5012         }

5014         /* XXX bob: given what modify_cobject does this seems unnecessary */
5015         fcp->c_attr.va_mask = AT_ALL;
5016         error = VOP_GETATTR(fcp->c_backvp, &fcp->c_attr, 0, cr, NULL);
5017         mutex_exit(&fcp->c_statelock);
5018 out:
5019         if (backvp)
5020                 VN_RELE(backvp);

5022         return (error);
5023 }

5025 static int
5026 cachefs_link_disconnected(vnode_t *tdvp, vnode_t *fvp, char *tnm,
5027     cred_t *cr)
5028 {
5029         cnode_t *tdcp = VTOC(tdvp);
5030         cnode_t *fcp = VTOC(fvp);
5031         fscache_t *fscp = VFS_TO_FSCACHE(tdvp->v_vfsp);
5032         int error = 0;
5033         timestruc_t current_time;
5034         off_t commit = 0;

5036         if (fvp->v_type == VDIR && secpolicy_fs_linkdir(cr, fvp->v_vfsp) != 0 ||
5037             fcp->c_attr.va_uid != crgetuid(cr) && secpolicy_basic_link(cr) != 0)
5038                 return (EPERM);

5040         if (CFS_ISFS_WRITE_AROUND(fscp))
5041                 return (ETIMEDOUT);

5043         if (fcp->c_metadata.md_flags & MD_NEEDATTRS)
5044                 return (ETIMEDOUT);

5046         mutex_enter(&tdcp->c_statelock);

5048         /* check permissions */
5049         if (error = cachefs_access_local(tdcp, (VEXEC|VWRITE), cr)) {
5050                 mutex_exit(&tdcp->c_statelock);
5051                 goto out;
5052         }

5054         /* the directory front file must be populated */
5055         if ((tdcp->c_metadata.md_flags & MD_POPULATED) == 0) {
5056                 error = ETIMEDOUT;
5057                 mutex_exit(&tdcp->c_statelock);
5058                 goto out;
5059         }

5061         /* make sure tnm does not already exist in the directory */
5062         error = cachefs_dir_look(tdcp, tnm, NULL, NULL, NULL, NULL);
5063         if (error == ENOTDIR) {
5064                 error = ETIMEDOUT;
5065                 mutex_exit(&tdcp->c_statelock);
5066                 goto out;
5067         }
5068         if (error != ENOENT) {
5069                 error = EEXIST;
5070                 mutex_exit(&tdcp->c_statelock);
5071                 goto out;
5072         }

5074         mutex_enter(&fcp->c_statelock);

5076         /* create a mapping for the file if necessary */
5077         if ((fcp->c_metadata.md_flags & MD_MAPPING) == 0) {
```



```
5078                 error = cachefs_dlog_cidmap(fscp);
5079                 if (error) {
5080                         mutex_exit(&fcp->c_statelock);
5081                         mutex_exit(&tdcp->c_statelock);
5082                         error = ENOSPC;
5083                         goto out;
5084                 }
5085                 fcp->c_metadata.md_flags |= MD_MAPPING;
5086                 fcp->c_flags |= CN_UPDATED;
5087         }

5089         /* mark file as modified */
5090         if (cachefs_modified_alloc(fcp)) {
5091                 mutex_exit(&fcp->c_statelock);
5092                 mutex_exit(&tdcp->c_statelock);
5093                 error = ENOSPC;
5094                 goto out;
5095         }
5096         mutex_exit(&fcp->c_statelock);

5098         /* log the operation */
5099         commit = cachefs_dlog_link(fscp, tdcp, tnm, fcp, cr);
5100         if (commit == 0) {
5101                 mutex_exit(&tdcp->c_statelock);
5102                 error = ENOSPC;
5103                 goto out;
5104         }

5106         gethrestime(&current_time);

5108         /* make the new link */
5109         cachefs_modified(tdcp);
5110         error = cachefs_dir_enter(tdcp, tnm, &fcp->c_cookie,
5111             &fcp->c_id, SM_ASYNC);
5112         if (error) {
5113                 error = 0;
5114                 mutex_exit(&tdcp->c_statelock);
5115                 goto out;
5116         }

5118         /* Update mtime/ctime of parent dir */
5119         tdcp->c_metadata.md_localmtime = current_time;
5120         tdcp->c_metadata.md_localctime = current_time;
5121         tdcp->c_metadata.md_flags |= MD_LOCALCTIME | MD_LOCALMTIME;
5122         tdcp->c_flags |= CN_UPDATED;
5123         mutex_exit(&tdcp->c_statelock);

5125         /* update the file we linked to */
5126         mutex_enter(&fcp->c_statelock);
5127         fcp->c_attr.va_nlink++;
5128         fcp->c_metadata.md_localctime = current_time;
5129         fcp->c_metadata.md_flags |= MD_LOCALCTIME;
5130         fcp->c_flags |= CN_UPDATED;
5131         mutex_exit(&fcp->c_statelock);

5133 out:
5134         if (commit) {
5135                 /* commit the log entry */
5136                 if (cachefs_dlog_commit(fscp, commit, error)) {
5137                         /*EMPTY*/
5138                         /* XXX bob: fix on panic */
5139                 }
5140         }

5142         return (error);
5143 }
```

```
5145 /*
5146  * Serialize all renames in CFS, to avoid deadlocks - We have to hold two
5147  * cnodes atomically.
5148  */
5149 kmutex_t cachefs_rename_lock;

5151 /*ARGSUSED*/
5152 static int
5153 cachefs_rename(vnode_t *odvp, char *onm, vnode_t *ndvp,
5154     char *nnm, cred_t *cr, caller_context_t *ct, int flags)
5155 {
5156         fscache_t *fscp = C_TO_FSCACHE(VTOC(odvp));
5157         cachefscache_t *cachep = fscp->fs_cache;
5158         int error = 0;
5159         int held = 0;
5160         int connected = 0;
5161         vnode_t *delvp = NULL;
5162         vnode_t *tvp = NULL;
5163         int vfslock = 0;
5164         struct vnode *realvp;

5166         if (getzoneid() != GLOBAL_ZONEID)
5167                 return (EPERM);

5169         if (VOP_REALVP(ndvp, &realvp, ct) == 0)
5170                 ndvp = realvp;

5172         /*
5173          * if the fs NOFILL or NOCACHE flags are on, then the old and new
5174          * directory cnodes better indicate NOCACHE mode as well.
5175          */
5176         ASSERT(
5177             (fscp->fs_cache->c_flags & (CACHE_NOFILL | CACHE_NOCACHE)) == 0 ||
5178             ((VTOC(odvp)->c_flags & CN_NOCACHE) &&
5179             (VTOC(ndvp)->c_flags & CN_NOCACHE)));

5181         /*
5182          * Cachefs only provides pass-through support for NFSv4,
5183          * and all vnode operations are passed through to the
5184          * back file system. For NFSv4 pass-through to work, only
5185          * connected operation is supported, the cnode backvp must
5186          * exist, and cachefs optional (eg., disconnectable) flags
5187          * are turned off. Assert these conditions to ensure that
5188          * the backfilesystem is called for the rename operation.
5189          */
5190         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
5191         CFS_BACKFS_NFSV4_ASSERT_CNODE(VTOC(odvp));
5192         CFS_BACKFS_NFSV4_ASSERT_CNODE(VTOC(ndvp));

5194         for (;;) {
5195                 if (vfslock) {
5196                         vn_vfsunlock(delvp);
5197                         vfslock = 0;
5198                 }
5199                 if (delvp) {
5200                         VN_RELE(delvp);
5201                         delvp = NULL;
5202                 }

5204                 /* get (or renew) access to the file system */
5205                 if (held) {
5206                         /* Won't loop for NFSv4 connected support */
5207                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
5208                         cachefs_cd_release(fscp);
5209                         held = 0;
```



```
5210                 }
5211                 error = cachefs_cd_access(fscp, connected, 1);
5212                 if (error)
5213                         break;
5214                 held = 1;

5216                 /* sanity check */
5217                 if ((odvp->v_type != VDIR) || (ndvp->v_type != VDIR)) {
5218                         error = EINVAL;
5219                         break;
5220                 }

5222                 /* cannot rename from or to . or .. */
5223                 if (strcmp(onm, ".") == 0 || strcmp(onm, "..") == 0 ||
5224                     strcmp(nnm, ".") == 0 || strcmp(nnm, "..") == 0) {
5225                         error = EINVAL;
5226                         break;
5227                 }

5229                 if (odvp != ndvp) {
5230                         /*
5231                          * if moving a directory, its notion
5232                          * of ".." will change
5233                          */
5234                         error = cachefs_lookup_common(odvp, onm, &tvp,
5235                             NULL, 0, NULL, cr);
5236                         if (error == 0) {
5237                                 ASSERT(tvp != NULL);
5238                                 if (tvp->v_type == VDIR) {
5239                                         cnode_t *cp = VTOC(tvp);

5241                                         dnlc_remove(tvp, "..");

5243                                         mutex_enter(&cp->c_statelock);
5244                                         CFSOP_MODIFY_COBJECT(fscp, cp, cr);
5245                                         mutex_exit(&cp->c_statelock);
5246                                 }
5247                         } else {
5248                                 tvp = NULL;
5249                                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
5250                                         if (CFS_TIMEOUT(fscp, error)) {
5251                                                 cachefs_cd_release(fscp);
5252                                                 held = 0;
5253                                                 cachefs_cd_timedout(fscp);
5254                                                 connected = 0;
5255                                                 continue;
5256                                         }
5257                                 } else {
5258                                         if (CFS_TIMEOUT(fscp, error)) {
5259                                                 connected = 1;
5260                                                 continue;
5261                                         }
5262                                 }
5263                                 break;
5264                         }
5265                 }

5267                 /* get the cnode if file being deleted */
5268                 error = cachefs_lookup_common(ndvp, nnm, &delvp, NULL, 0,
5269                     NULL, cr);
5270                 if (error) {
5271                         delvp = NULL;
5272                         if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
5273                                 if (CFS_TIMEOUT(fscp, error)) {
5274                                         cachefs_cd_release(fscp);
5275                                         held = 0;
```

```
5276                                    cachefs_cd_timedout(fscp);
5277                                    connected = 0;
5278                                    continue;
5279                            }
5280                    } else {
5281                            if (CFS_TIMEOUT(fscp, error)) {
5282                                    connected = 1;
5283                                    continue;
5284                            }
5285                    }
5286                    if (error != ENOENT)
5287                            break;
5288            }

5290            if (delvp && delvp->v_type == VDIR) {
5291                    /* see ufs_dirremove for why this is done, mount race */
5292                    if (vn_vfswlock(delvp)) {
5293                            error = EBUSY;
5294                            break;
5295                    }
5296                    vfslock = 1;
5297                    if (vn_mountedvfs(delvp) != NULL) {
5298                            error = EBUSY;
5299                            break;
5300                    }
5301            }

5303            if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
5304                    error = cachefs_rename_connected(odvp, onm,
5305                        ndvp, nnm, cr, delvp);
5306                    if (CFS_TIMEOUT(fscp, error)) {
5307                            cachefs_cd_release(fscp);
5308                            held = 0;
5309                            cachefs_cd_timedout(fscp);
5310                            connected = 0;
5311                            continue;
5312                    }
5313            } else {
5314                    error = cachefs_rename_disconnected(odvp, onm,
5315                        ndvp, nnm, cr, delvp);
5316                    if (CFS_TIMEOUT(fscp, error)) {
5317                            connected = 1;
5318                            continue;
5319                    }
5320            }
5321            break;
5322    }

5324    if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_RENAME)) {
5325            struct fid gone;

5327            bzero(&gone, sizeof (gone));
5328            gone.fid_len = MAXFIDSZ;
5329            if (delvp != NULL)
5330                    (void) VOP_FID(delvp, &gone, ct);

5332            cachefs_log_rename(cachep, error, fscp->fs_cfsvfsp,
5333                &gone, 0, (delvp != NULL), crgetuid(cr));
5334    }

5336    if (held)
5337            cachefs_cd_release(fscp);

5339    if (vfslock)
5340            vn_vfsunlock(delvp);
```



```
5342    if (delvp)
5343            VN_RELE(delvp);
5344    if (tvp)
5345            VN_RELE(tvp);

5347 #ifdef CFS_CD_DEBUG
5348    ASSERT((curthread->t_flag & T_CD_HELD) == 0);
5349 #endif
5350    return (error);
5351 }

5353 static int
5354 cachefs_rename_connected(vnode_t *odvp, char *onm, vnode_t *ndvp,
5355     char *nnm, cred_t *cr, vnode_t *delvp)
5356 {
5357    cnode_t *odcp = VTOC(odvp);
5358    cnode_t *ndcp = VTOC(ndvp);
5359    vnode_t *revp = NULL;
5360    cnode_t *recp;
5361    cnode_t *delcp;
5362    fscache_t *fscp = C_TO_FSCACHE(odcp);
5363    int error = 0;
5364    struct fid cookie;
5365    struct fid *cookiep;
5366    cfs_cid_t cid;
5367    int gotdirent;

5369    /* find the file we are renaming */
5370    error = cachefs_lookup_common(odvp, onm, &revp, NULL, 0, NULL, cr);
5371    if (error)
5372            return (error);
5373    recp = VTOC(revp);

5375    /*
5376     * To avoid deadlock, we acquire this global rename lock before
5377     * we try to get the locks for the source and target directories.
5378     */
5379    mutex_enter(&cachefs_rename_lock);
5380    rw_enter(&odcp->c_rwlock, RW_WRITER);
5381    if (odcp != ndcp) {
5382            rw_enter(&ndcp->c_rwlock, RW_WRITER);
5383    }
5384    mutex_exit(&cachefs_rename_lock);

5386    ASSERT((odcp->c_flags & CN_ASYNC_POP_WORKING) == 0);
5387    ASSERT((ndcp->c_flags & CN_ASYNC_POP_WORKING) == 0);

5389    mutex_enter(&odcp->c_statelock);
5390    if (odcp->c_backvp == NULL) {
5391            error = cachefs_getbackvp(fscp, odcp);
5392            if (error) {
5393                    mutex_exit(&odcp->c_statelock);
5394                    goto out;
5395            }
5396    }

5398    error = CFSOP_CHECK_COBJECT(fscp, odcp, 0, cr);
5399    if (error) {
5400            mutex_exit(&odcp->c_statelock);
5401            goto out;
5402    }
5403    mutex_exit(&odcp->c_statelock);

5405    if (odcp != ndcp) {
5406            mutex_enter(&ndcp->c_statelock);
5407            if (ndcp->c_backvp == NULL) {
```

```
5408                        error = cachefs_getbackvp(fscp, ndcp);
5409                        if (error) {
5410                                mutex_exit(&ndcp->c_statelock);
5411                                goto out;
5412                        }
5413                }

5415                error = CFSOP_CHECK_COBJECT(fscp, ndcp, 0, cr);
5416                if (error) {
5417                        mutex_exit(&ndcp->c_statelock);
5418                        goto out;
5419                }
5420                mutex_exit(&ndcp->c_statelock);
5421        }

5423        /* if a file is being deleted because of this rename */
5424        if (delvp) {
5425                /* if src and dest file are same */
5426                if (delvp == revp) {
5427                        error = 0;
5428                        goto out;
5429                }

5431                /*
5432                 * If the cnode is active, make a link to the file
5433                 * so operations on the file will continue.
5434                 */
5435                dnlc_purge_vp(delvp);
5436                delcp = VTOC(delvp);
5437                if ((delvp->v_type != VDIR) &&
5438                    !((delvp->v_count == 1) ||
5439                    ((delvp->v_count == 2) && delcp->c_ipending))) {
5440                        error = cachefs_remove_dolink(ndvp, delvp, nnm, cr);
5441                        if (error)
5442                                goto out;
5443                }
5444        }

5446        /* do the rename on the back fs */
5447        CFS_DPRINT_BACKFS_NFSV4(fscp,
5448            ("cachefs_rename (nfsv4): odcp %p, odbackvp %p, "
5449            " ndcp %p, ndbackvp %p, onm %s, nnm %s\n",
5450            odcp, odcp->c_backvp, ndcp, ndcp->c_backvp, onm, nnm));
5451        error = VOP_RENAME(odcp->c_backvp, onm, ndcp->c_backvp, nnm, cr, NULL,
5452            0);
5453        if (error)
5454                goto out;

5456        /* purge mappings to file in the old directory */
5457        dnlc_purge_vp(odvp);

5459        /* purge mappings in the new dir if we deleted a file */
5460        if (delvp && (odvp != ndvp))
5461                dnlc_purge_vp(ndvp);

5463        /* update the file we just deleted */
5464        if (delvp) {
5465                mutex_enter(&delcp->c_statelock);
5466                if (delcp->c_attr.va_nlink == 1) {
5467                        delcp->c_flags |= CN_DESTROY;
5468                } else {
5469                        delcp->c_flags |= CN_UPDATED;
5470                }
5471                delcp->c_attr.va_nlink--;
5472                CFSOP_MODIFY_COBJECT(fscp, delcp, cr);
5473                mutex_exit(&delcp->c_statelock);
```



```
5474        }

5476        /* find the entry in the old directory */
5477        mutex_enter(&odcp->c_statelock);
5478        gotdirent = 0;
5479        cookiep = NULL;
5480        if (CFS_ISFS_NONSHARED(fscp) &&
5481            (odcp->c_metadata.md_flags & MD_POPULATED)) {
5482                error = cachefs_dir_look(odcp, onm, &cookie,
5483                    NULL, NULL, &cid);
5484                if (error == 0 || error == EINVAL) {
5485                        gotdirent = 1;
5486                        if (error == 0)
5487                                cookiep = &cookie;
5488                } else {
5489                        cachefs_inval_object(odcp);
5490                }
5491        }
5492        error = 0;

5494        /* remove the directory entry from the old directory */
5495        if (gotdirent) {
5496                error = cachefs_dir_rmentry(odcp, onm);
5497                if (error) {
5498                        cachefs_nocache(odcp);
5499                        error = 0;
5500                }
5501        }
5502        CFSOP_MODIFY_COBJECT(fscp, odcp, cr);
5503        mutex_exit(&odcp->c_statelock);

5505        /* install the directory entry in the new directory */
5506        mutex_enter(&ndcp->c_statelock);
5507        if (CFS_ISFS_NONSHARED(fscp) &&
5508            (ndcp->c_metadata.md_flags & MD_POPULATED)) {
5509                error = 1;
5510                if (gotdirent) {
5511                        ASSERT(cid.cid_fileno != 0);
5512                        error = 0;
5513                        if (delvp) {
5514                                error = cachefs_dir_rmentry(ndcp, nnm);
5515                        }
5516                        if (error == 0) {
5517                                error = cachefs_dir_enter(ndcp, nnm, cookiep,
5518                                    &cid, SM_ASYNC);
5519                        }
5520                }
5521                if (error) {
5522                        cachefs_nocache(ndcp);
5523                        error = 0;
5524                }
5525        }
5526        if (odcp != ndcp)
5527                CFSOP_MODIFY_COBJECT(fscp, ndcp, cr);
5528        mutex_exit(&ndcp->c_statelock);

5530        /* ctime of renamed file has changed */
5531        mutex_enter(&recp->c_statelock);
5532        CFSOP_MODIFY_COBJECT(fscp, recp, cr);
5533        mutex_exit(&recp->c_statelock);

5535 out:
5536        if (odcp != ndcp)
5537                rw_exit(&ndcp->c_rwlock);
5538        rw_exit(&odcp->c_rwlock);
```

```
5540         VN_RELE(revp);

5542         return (error);
5543 }

5545 static int
5546 cachefs_rename_disconnected(vnode_t *odvp, char *onm, vnode_t *ndvp,
5547     char *nnm, cred_t *cr, vnode_t *delvp)
5548 {
5549         cnode_t *odcp = VTOC(odvp);
5550         cnode_t *ndcp = VTOC(ndvp);
5551         cnode_t *delcp = NULL;
5552         vnode_t *revp = NULL;
5553         cnode_t *recp;
5554         fscache_t *fscp = C_TO_FSCACHE(odcp);
5555         int error = 0;
5556         struct fid cookie;
5557         struct fid *cookiep;
5558         cfs_cid_t cid;
5559         off_t commit = 0;
5560         timestruc_t current_time;

5562         if (CFS_ISFS_WRITE_AROUND(fscp))
5563                 return (ETIMEDOUT);

5565         /* find the file we are renaming */
5566         error = cachefs_lookup_common(odvp, onm, &revp, NULL, 0, NULL, cr);
5567         if (error)
5568                 return (error);
5569         recp = VTOC(revp);

5571         /*
5572          * To avoid deadlock, we acquire this global rename lock before
5573          * we try to get the locks for the source and target directories.
5574          */
5575         mutex_enter(&cachefs_rename_lock);
5576         rw_enter(&odcp->c_rwlock, RW_WRITER);
5577         if (odcp != ndcp) {
5578                 rw_enter(&ndcp->c_rwlock, RW_WRITER);
5579         }
5580         mutex_exit(&cachefs_rename_lock);

5582         if (recp->c_metadata.md_flags & MD_NEEDATTRS) {
5583                 error = ETIMEDOUT;
5584                 goto out;
5585         }

5587         if ((recp->c_metadata.md_flags & MD_MAPPING) == 0) {
5588                 mutex_enter(&recp->c_statelock);
5589                 if ((recp->c_metadata.md_flags & MD_MAPPING) == 0) {
5590                         error = cachefs_dlog_cidmap(fscp);
5591                         if (error) {
5592                                 mutex_exit(&recp->c_statelock);
5593                                 error = ENOSPC;
5594                                 goto out;
5595                         }
5596                         recp->c_metadata.md_flags |= MD_MAPPING;
5597                         recp->c_flags |= CN_UPDATED;
5598                 }
5599                 mutex_exit(&recp->c_statelock);
5600         }

5602         /* check permissions */
5603         /* XXX clean up this mutex junk sometime */
5604         mutex_enter(&odcp->c_statelock);
5605         error = cachefs_access_local(odcp, (VEXEC|VWRITE), cr);
```



```
5606         mutex_exit(&odcp->c_statelock);
5607         if (error != 0)
5608                 goto out;
5609         mutex_enter(&ndcp->c_statelock);
5610         error = cachefs_access_local(ndcp, (VEXEC|VWRITE), cr);
5611         mutex_exit(&ndcp->c_statelock);
5612         if (error != 0)
5613                 goto out;
5614         mutex_enter(&odcp->c_statelock);
5615         error = cachefs_stickyrmchk(odcp, recp, cr);
5616         mutex_exit(&odcp->c_statelock);
5617         if (error != 0)
5618                 goto out;

5620         /* dirs must be populated */
5621         if (((odcp->c_metadata.md_flags & MD_POPULATED) == 0) ||
5622             ((ndcp->c_metadata.md_flags & MD_POPULATED) == 0)) {
5623                 error = ETIMEDOUT;
5624                 goto out;
5625         }

5627         /* for now do not allow moving dirs because could cause cycles */
5628         if ((((revp->v_type == VDIR) && (odvp != ndvp))) ||
5629             (revp == odvp)) {
5630                 error = ETIMEDOUT;
5631                 goto out;
5632         }

5634         /* if a file is being deleted because of this rename */
5635         if (delvp) {
5636                 delcp = VTOC(delvp);

5638                 /* if src and dest file are the same */
5639                 if (delvp == revp) {
5640                         error = 0;
5641                         goto out;
5642                 }

5644                 if (delcp->c_metadata.md_flags & MD_NEEDATTRS) {
5645                         error = ETIMEDOUT;
5646                         goto out;
5647                 }

5649                 /* if there are hard links to this file */
5650                 if (delcp->c_attr.va_nlink > 1) {
5651                         mutex_enter(&delcp->c_statelock);
5652                         if (cachefs_modified_alloc(delcp)) {
5653                                 mutex_exit(&delcp->c_statelock);
5654                                 error = ENOSPC;
5655                                 goto out;
5656                         }

5658                         if ((delcp->c_metadata.md_flags & MD_MAPPING) == 0) {
5659                                 error = cachefs_dlog_cidmap(fscp);
5660                                 if (error) {
5661                                         mutex_exit(&delcp->c_statelock);
5662                                         error = ENOSPC;
5663                                         goto out;
5664                                 }
5665                                 delcp->c_metadata.md_flags |= MD_MAPPING;
5666                                 delcp->c_flags |= CN_UPDATED;
5667                         }
5668                         mutex_exit(&delcp->c_statelock);
5669                 }

5671                 /* make sure we can delete file */
```

```
5672                mutex_enter(&ndcp->c_statelock);
5673                error = cachefs_stickyrmchk(ndcp, delcp, cr);
5674                mutex_exit(&ndcp->c_statelock);
5675                if (error != 0)
5676                        goto out;

5678                /*
5679                 * If the cnode is active, make a link to the file
5680                 * so operations on the file will continue.
5681                 */
5682                dnlc_purge_vp(delvp);
5683                if ((delvp->v_type != VDIR) &&
5684                    !((delvp->v_count == 1) ||
5685                    ((delvp->v_count == 2) && delcp->c_ipending))) {
5686                        error = cachefs_remove_dolink(ndvp, delvp, nnm, cr);
5687                        if (error)
5688                                goto out;
5689                }
5690        }

5692        /* purge mappings to file in the old directory */
5693        dnlc_purge_vp(odvp);

5695        /* purge mappings in the new dir if we deleted a file */
5696        if (delvp && (odvp != ndvp))
5697                dnlc_purge_vp(ndvp);

5699        /* find the entry in the old directory */
5700        mutex_enter(&odcp->c_statelock);
5701        if ((odcp->c_metadata.md_flags & MD_POPULATED) == 0) {
5702                mutex_exit(&odcp->c_statelock);
5703                error = ETIMEDOUT;
5704                goto out;
5705        }
5706        cookiep = NULL;
5707        error = cachefs_dir_look(odcp, onm, &cookie, NULL, NULL, &cid);
5708        if (error == 0 || error == EINVAL) {
5709                if (error == 0)
5710                        cookiep = &cookie;
5711        } else {
5712                mutex_exit(&odcp->c_statelock);
5713                if (error == ENOTDIR)
5714                        error = ETIMEDOUT;
5715                goto out;
5716        }
5717        error = 0;

5719        /* write the log entry */
5720        commit = cachefs_dlog_rename(fscp, odcp, onm, ndcp, nnm, cr,
5721            recp, delcp);
5722        if (commit == 0) {
5723                mutex_exit(&odcp->c_statelock);
5724                error = ENOSPC;
5725                goto out;
5726        }

5728        /* remove the directory entry from the old directory */
5729        cachefs_modified(odcp);
5730        error = cachefs_dir_rmentry(odcp, onm);
5731        if (error) {
5732                mutex_exit(&odcp->c_statelock);
5733                if (error == ENOTDIR)
5734                        error = ETIMEDOUT;
5735                goto out;
5736        }
5737        mutex_exit(&odcp->c_statelock);
```



```
5739        /* install the directory entry in the new directory */
5740        mutex_enter(&ndcp->c_statelock);
5741        error = ENOTDIR;
5742        if (ndcp->c_metadata.md_flags & MD_POPULATED) {
5743                ASSERT(cid.cid_fileno != 0);
5744                cachefs_modified(ndcp);
5745                error = 0;
5746                if (delvp) {
5747                        error = cachefs_dir_rmentry(ndcp, nnm);
5748                }
5749                if (error == 0) {
5750                        error = cachefs_dir_enter(ndcp, nnm, cookiep,
5751                            &cid, SM_ASYNC);
5752                }
5753        }
5754        if (error) {
5755                cachefs_nocache(ndcp);
5756                mutex_exit(&ndcp->c_statelock);
5757                mutex_enter(&odcp->c_statelock);
5758                cachefs_nocache(odcp);
5759                mutex_exit(&odcp->c_statelock);
5760                if (error == ENOTDIR)
5761                        error = ETIMEDOUT;
5762                goto out;
5763        }
5764        mutex_exit(&ndcp->c_statelock);

5766        gethrestime(&current_time);

5768        /* update the file we just deleted */
5769        if (delvp) {
5770                mutex_enter(&delcp->c_statelock);
5771                delcp->c_attr.va_nlink--;
5772                delcp->c_metadata.md_localctime = current_time;
5773                delcp->c_metadata.md_flags |= MD_LOCALCTIME;
5774                if (delcp->c_attr.va_nlink == 0) {
5775                        delcp->c_flags |= CN_DESTROY;
5776                } else {
5777                        delcp->c_flags |= CN_UPDATED;
5778                }
5779                mutex_exit(&delcp->c_statelock);
5780        }

5782        /* update the file we renamed */
5783        mutex_enter(&recp->c_statelock);
5784        recp->c_metadata.md_localctime = current_time;
5785        recp->c_metadata.md_flags |= MD_LOCALCTIME;
5786        recp->c_flags |= CN_UPDATED;
5787        mutex_exit(&recp->c_statelock);

5789        /* update the source directory */
5790        mutex_enter(&odcp->c_statelock);
5791        odcp->c_metadata.md_localctime = current_time;
5792        odcp->c_metadata.md_localmtime = current_time;
5793        odcp->c_metadata.md_flags |= MD_LOCALCTIME | MD_LOCALMTIME;
5794        odcp->c_flags |= CN_UPDATED;
5795        mutex_exit(&odcp->c_statelock);

5797        /* update the destination directory */
5798        if (odcp != ndcp) {
5799                mutex_enter(&ndcp->c_statelock);
5800                ndcp->c_metadata.md_localctime = current_time;
5801                ndcp->c_metadata.md_localmtime = current_time;
5802                ndcp->c_metadata.md_flags |= MD_LOCALCTIME | MD_LOCALMTIME;
5803                ndcp->c_flags |= CN_UPDATED;
```

```
5804                    mutex_exit(&ndcp->c_statelock);
5805            }

5807    out:
5808            if (commit) {
5809                    /* commit the log entry */
5810                    if (cachefs_dlog_commit(fscp, commit, error)) {
5811                            /*EMPTY*/
5812                            /* XXX bob: fix on panic */
5813                    }
5814            }

5816            if (odcp != ndcp)
5817                    rw_exit(&ndcp->c_rwlock);
5818            rw_exit(&odcp->c_rwlock);

5820            VN_RELE(revp);

5822            return (error);
5823    }

5825    /*ARGSUSED*/
5826    static int
5827    cachefs_mkdir(vnode_t *dvp, char *nm, vattr_t *vap, vnode_t **vpp,
5828        cred_t *cr, caller_context_t *ct, int flags, vsecattr_t *vsecp)
5829    {
5830            cnode_t *dcp = VTOC(dvp);
5831            fscache_t *fscp = C_TO_FSCACHE(dcp);
5832            cachefscache_t *cachep = fscp->fs_cache;
5833            int error = 0;
5834            int held = 0;
5835            int connected = 0;

5837    #ifdef CFSDEBUG
5838            CFS_DEBUG(CFSDEBUG_VOPS)
5839                    printf("cachefs_mkdir: ENTER dvp %p\n", (void *)dvp);
5840    #endif

5842            if (getzoneid() != GLOBAL_ZONEID) {
5843                    error = EPERM;
5844                    goto out;
5845            }

5847            if (fscp->fs_cache->c_flags & (CACHE_NOFILL | CACHE_NOCACHE))
5848                    ASSERT(dcp->c_flags & CN_NOCACHE);

5850            /*
5851             * Cachefs only provides pass-through support for NFSv4,
5852             * and all vnode operations are passed through to the
5853             * back file system. For NFSv4 pass-through to work, only
5854             * connected operation is supported, the cnode backvp must
5855             * exist, and cachefs optional (eg., disconnectable) flags
5856             * are turned off. Assert these conditions to ensure that
5857             * the backfilesystem is called for the mkdir operation.
5858             */
5859            CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
5860            CFS_BACKFS_NFSV4_ASSERT_CNODE(dcp);

5862            for (;;) {
5863                    /* get (or renew) access to the file system */
5864                    if (held) {
5865                            /* Won't loop with NFSv4 connected behavior */
5866                            ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
5867                            rw_exit(&dcp->c_rwlock);
5868                            cachefs_cd_release(fscp);
5869                            held = 0;
```



```
5870                    }
5871                    error = cachefs_cd_access(fscp, connected, 1);
5872                    if (error)
5873                            break;
5874                    rw_enter(&dcp->c_rwlock, RW_WRITER);
5875                    held = 1;

5877                    if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
5878                            error = cachefs_mkdir_connected(dvp, nm, vap,
5879                                vpp, cr);
5880                            if (CFS_TIMEOUT(fscp, error)) {
5881                                    rw_exit(&dcp->c_rwlock);
5882                                    cachefs_cd_release(fscp);
5883                                    held = 0;
5884                                    cachefs_cd_timedout(fscp);
5885                                    connected = 0;
5886                                    continue;
5887                            }
5888                    } else {
5889                            error = cachefs_mkdir_disconnected(dvp, nm, vap,
5890                                vpp, cr);
5891                            if (CFS_TIMEOUT(fscp, error)) {
5892                                    connected = 1;
5893                                    continue;
5894                            }
5895                    }
5896                    break;
5897            }

5899            if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_MKDIR)) {
5900                    fid_t *fidp = NULL;
5901                    ino64_t fileno = 0;
5902                    cnode_t *cp = NULL;
5903                    if (error == 0)
5904                            cp = VTOC(*vpp);

5906                    if (cp != NULL) {
5907                            fidp = &cp->c_metadata.md_cookie;
5908                            fileno = cp->c_id.cid_fileno;
5909                    }

5911                    cachefs_log_mkdir(cachep, error, fscp->fs_cfsvfsp,
5912                        fidp, fileno, crgetuid(cr));
5913            }

5915            if (held) {
5916                    rw_exit(&dcp->c_rwlock);
5917                    cachefs_cd_release(fscp);
5918            }
5919            if (error == 0 && CFS_ISFS_NONSHARED(fscp))
5920                    (void) cachefs_pack(dvp, nm, cr);

5922    #ifdef CFS_CD_DEBUG
5923            ASSERT((curthread->t_flag & T_CD_HELD) == 0);
5924    #endif
5925    out:
5926    #ifdef CFSDEBUG
5927            CFS_DEBUG(CFSDEBUG_VOPS)
5928                    printf("cachefs_mkdir: EXIT error = %d\n", error);
5929    #endif
5930            return (error);
5931    }

5933    static int
5934    cachefs_mkdir_connected(vnode_t *dvp, char *nm, vattr_t *vap,
5935        vnode_t **vpp, cred_t *cr)
```

```
{
        cnode_t *newcp = NULL, *dcp = VTOC(dvp);
        struct vnode *vp = NULL;
        int error = 0;
        fscache_t *fscp = C_TO_FSCACHE(dcp);
        struct fid cookie;
        struct vattr attr;
        cfs_cid_t cid, dircid;
        uint32_t valid_fid;

        if (fscp->fs_cache->c_flags & (CACHE_NOFILL | CACHE_NOCACHE))
                ASSERT(dcp->c_flags & CN_NOCACHE);

        mutex_enter(&dcp->c_statelock);

        /* get backvp of dir */
        if (dcp->c_backvp == NULL) {
                error = cachefs_getbackvp(fscp, dcp);
                if (error) {
                        mutex_exit(&dcp->c_statelock);
                        goto out;
                }
        }

        /* consistency check the directory */
        error = CFSOP_CHECK_COBJECT(fscp, dcp, 0, cr);
        if (error) {
                mutex_exit(&dcp->c_statelock);
                goto out;
        }
        dircid = dcp->c_id;

        /* make the dir on the back fs */
        CFS_DPRINT_BACKFS_NFSV4(fscp,
            ("cachefs_mkdir (nfsv4): dcp %p, dbackvp %p, "
            "name %s\n", dcp, dcp->c_backvp, nm));
        error = VOP_MKDIR(dcp->c_backvp, nm, vap, &vp, cr, NULL, 0, NULL);
        mutex_exit(&dcp->c_statelock);
        if (error) {
                goto out;
        }

        /* get the cookie and make the cnode */
        attr.va_mask = AT_ALL;
        valid_fid = (CFS_ISFS_BACKFS_NFSV4(fscp) ? FALSE : TRUE);
        error = cachefs_getcookie(vp, &cookie, &attr, cr, valid_fid);
        if (error) {
                goto out;
        }
        cid.cid_flags = 0;
        cid.cid_fileno = attr.va_nodeid;
        error = cachefs_cnode_make(&cid, fscp, (valid_fid ? &cookie : NULL),
            &attr, vp, cr, 0, &newcp);
        if (error) {
                goto out;
        }
        ASSERT(CTOV(newcp)->v_type == VDIR);
        *vpp = CTOV(newcp);

        /* if the dir is populated, add the new entry */
        mutex_enter(&dcp->c_statelock);
        if (CFS_ISFS_NONSHARED(fscp) &&
            (dcp->c_metadata.md_flags & MD_POPULATED)) {
                error = cachefs_dir_enter(dcp, nm, &cookie, &newcp->c_id,
                    SM_ASYNC);
                if (error) {
```



```
                        cachefs_nocache(dcp);
                        error = 0;
                }
        }
        dcp->c_attr.va_nlink++;
        dcp->c_flags |= CN_UPDATED;
        CFSOP_MODIFY_COBJECT(fscp, dcp, cr);
        mutex_exit(&dcp->c_statelock);

        /* XXX bob: should we do a filldir here? or just add . and .. */
        /* maybe should kick off an async filldir so caller does not wait */

        /* put the entry in the dnlc */
        if (cachefs_dnlc)
                dnlc_enter(dvp, nm, *vpp);

        /* save the fileno of the parent so can find the name */
        if (bcmp(&newcp->c_metadata.md_parent, &dircid,
            sizeof (cfs_cid_t)) != 0) {
                mutex_enter(&newcp->c_statelock);
                newcp->c_metadata.md_parent = dircid;
                newcp->c_flags |= CN_UPDATED;
                mutex_exit(&newcp->c_statelock);
        }
out:
        if (vp)
                VN_RELE(vp);

        return (error);
}

static int
cachefs_mkdir_disconnected(vnode_t *dvp, char *nm, vattr_t *vap,
    vnode_t **vpp, cred_t *cr)
{
        cnode_t *dcp = VTOC(dvp);
        fscache_t *fscp = C_TO_FSCACHE(dcp);
        int error;
        cnode_t *newcp = NULL;
        struct vattr va;
        timestruc_t current_time;
        off_t commit = 0;
        char *s;
        int namlen;

        /* don't allow '/' characters in pathname component */
        for (s = nm, namlen = 0; *s; s++, namlen++)
                if (*s == '/')
                        return (EACCES);
        if (namlen == 0)
                return (EINVAL);

        if (CFS_ISFS_WRITE_AROUND(fscp))
                return (ETIMEDOUT);

        mutex_enter(&dcp->c_statelock);

        /* check permissions */
        if (error = cachefs_access_local(dcp, (VEXEC|VWRITE), cr)) {
                mutex_exit(&dcp->c_statelock);
                goto out;
        }

        /* the directory front file must be populated */
        if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
                error = ETIMEDOUT;
```

```
6068                    mutex_exit(&dcp->c_statelock);
6069                    goto out;
6070            }

6072            /* make sure nm does not already exist in the directory */
6073            error = cachefs_dir_look(dcp, nm, NULL, NULL, NULL, NULL);
6074            if (error == ENOTDIR) {
6075                    error = ETIMEDOUT;
6076                    mutex_exit(&dcp->c_statelock);
6077                    goto out;
6078            }
6079            if (error != ENOENT) {
6080                    error = EEXIST;
6081                    mutex_exit(&dcp->c_statelock);
6082                    goto out;
6083            }

6085            /* make up a reasonable set of attributes */
6086            cachefs_attr_setup(vap, &va, dcp, cr);
6087            va.va_type = VDIR;
6088            va.va_mode |= S_IFDIR;
6089            va.va_nlink = 2;

6091            mutex_exit(&dcp->c_statelock);

6093            /* create the cnode */
6094            error = cachefs_cnode_create(fscp, &va, 0, &newcp);
6095            if (error)
6096                    goto out;

6098            mutex_enter(&newcp->c_statelock);

6100            error = cachefs_dlog_cidmap(fscp);
6101            if (error) {
6102                    mutex_exit(&newcp->c_statelock);
6103                    goto out;
6104            }

6106            cachefs_creategid(dcp, newcp, vap, cr);
6107            mutex_enter(&dcp->c_statelock);
6108            cachefs_createacl(dcp, newcp);
6109            mutex_exit(&dcp->c_statelock);
6110            gethrestime(&current_time);
6111            newcp->c_metadata.md_vattr.va_atime = current_time;
6112            newcp->c_metadata.md_localctime = current_time;
6113            newcp->c_metadata.md_localmtime = current_time;
6114            newcp->c_metadata.md_flags |= MD_MAPPING | MD_LOCALMTIME |
6115                MD_LOCALCTIME;
6116            newcp->c_flags |= CN_UPDATED;

6118            /* make a front file for the new directory, add . and .. */
6119            error = cachefs_dir_new(dcp, newcp);
6120            if (error) {
6121                    mutex_exit(&newcp->c_statelock);
6122                    goto out;
6123            }
6124            cachefs_modified(newcp);

6126            /*
6127             * write the metadata now rather than waiting until
6128             * inactive so that if there's no space we can let
6129             * the caller know.
6130             */
6131            ASSERT(newcp->c_frontvp);
6132            ASSERT((newcp->c_filegrp->fg_flags & CFS_FG_ALLOC_ATTR) == 0);
6133            ASSERT((newcp->c_flags & CN_ALLOC_PENDING) == 0);
```



```
6134            error = filegrp_write_metadata(newcp->c_filegrp,
6135                &newcp->c_id, &newcp->c_metadata);
6136            if (error) {
6137                    mutex_exit(&newcp->c_statelock);
6138                    goto out;
6139            }
6140            mutex_exit(&newcp->c_statelock);

6142            /* log the operation */
6143            commit = cachefs_dlog_mkdir(fscp, dcp, newcp, nm, &va, cr);
6144            if (commit == 0) {
6145                    error = ENOSPC;
6146                    goto out;
6147            }

6149            mutex_enter(&dcp->c_statelock);

6151            /* make sure directory is still populated */
6152            if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
6153                    mutex_exit(&dcp->c_statelock);
6154                    error = ETIMEDOUT;
6155                    goto out;
6156            }
6157            cachefs_modified(dcp);

6159            /* enter the new file in the directory */
6160            error = cachefs_dir_enter(dcp, nm, &newcp->c_metadata.md_cookie,
6161                &newcp->c_id, SM_ASYNC);
6162            if (error) {
6163                    mutex_exit(&dcp->c_statelock);
6164                    goto out;
6165            }

6167            /* update parent dir times */
6168            dcp->c_metadata.md_localctime = current_time;
6169            dcp->c_metadata.md_localmtime = current_time;
6170            dcp->c_metadata.md_flags |= MD_LOCALCTIME | MD_LOCALMTIME;
6171            dcp->c_attr.va_nlink++;
6172            dcp->c_flags |= CN_UPDATED;
6173            mutex_exit(&dcp->c_statelock);

6175    out:
6176            if (commit) {
6177                    /* commit the log entry */
6178                    if (cachefs_dlog_commit(fscp, commit, error)) {
6179                            /*EMPTY*/
6180                            /* XXX bob: fix on panic */
6181                    }
6182            }
6183            if (error) {
6184                    if (newcp) {
6185                            mutex_enter(&newcp->c_statelock);
6186                            newcp->c_flags |= CN_DESTROY;
6187                            mutex_exit(&newcp->c_statelock);
6188                            VN_RELE(CTOV(newcp));
6189                    }
6190            } else {
6191                    *vpp = CTOV(newcp);
6192            }
6193            return (error);
6194    }

6196    /*ARGSUSED*/
6197    static int
6198    cachefs_rmdir(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
6199        caller_context_t *ct, int flags)
```

```
{
        cnode_t *dcp = VTOC(dvp);
        fscache_t *fscp = C_TO_FSCACHE(dcp);
        cachefscache_t *cachep = fscp->fs_cache;
        int error = 0;
        int held = 0;
        int connected = 0;
        size_t namlen;
        vnode_t *vp = NULL;
        int vfslock = 0;

#ifdef CFSDEBUG
        CFS_DEBUG(CFSDEBUG_VOPS)
                printf("cachefs_rmdir: ENTER vp %p\n", (void *)dvp);
#endif

        if (getzoneid() != GLOBAL_ZONEID) {
                error = EPERM;
                goto out;
        }

        if (fscp->fs_cache->c_flags & (CACHE_NOFILL | CACHE_NOCACHE))
                ASSERT(dcp->c_flags & CN_NOCACHE);

        /*
         * Cachefs only provides pass-through support for NFSv4,
         * and all vnode operations are passed through to the
         * back file system. For NFSv4 pass-through to work, only
         * connected operation is supported, the cnode backvp must
         * exist, and cachefs optional (eg., disconnectable) flags
         * are turned off. Assert these conditions to ensure that
         * the backfilesystem is called for the rmdir operation.
         */
        CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
        CFS_BACKFS_NFSV4_ASSERT_CNODE(dcp);

        for (;;) {
                if (vfslock) {
                        vn_vfsunlock(vp);
                        vfslock = 0;
                }
                if (vp) {
                        VN_RELE(vp);
                        vp = NULL;
                }

                /* get (or renew) access to the file system */
                if (held) {
                        /* Won't loop with NFSv4 connected behavior */
                        ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
                        cachefs_cd_release(fscp);
                        held = 0;
                }
                error = cachefs_cd_access(fscp, connected, 1);
                if (error)
                        break;
                held = 1;

                /* if disconnected, do some extra error checking */
                if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
                        /* check permissions */
                        mutex_enter(&dcp->c_statelock);
                        error = cachefs_access_local(dcp, (VEXEC|VWRITE), cr);
                        mutex_exit(&dcp->c_statelock);
                        if (CFS_TIMEOUT(fscp, error)) {
                                connected = 1;
```



```
                                continue;
                        }
                        if (error)
                                break;

                        namlen = strlen(nm);
                        if (namlen == 0) {
                                error = EINVAL;
                                break;
                        }

                        /* cannot remove . and .. */
                        if (nm[0] == '.') {
                                if (namlen == 1) {
                                        error = EINVAL;
                                        break;
                                } else if (namlen == 2 && nm[1] == '.') {
                                        error = EEXIST;
                                        break;
                                }
                        }

                }

                /* get the cnode of the dir to remove */
                error = cachefs_lookup_common(dvp, nm, &vp, NULL, 0, NULL, cr);
                if (error) {
                        if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
                                if (CFS_TIMEOUT(fscp, error)) {
                                        cachefs_cd_release(fscp);
                                        held = 0;
                                        cachefs_cd_timedout(fscp);
                                        connected = 0;
                                        continue;
                                }
                        } else {
                                if (CFS_TIMEOUT(fscp, error)) {
                                        connected = 1;
                                        continue;
                                }
                        }
                        break;
                }

                /* must be a dir */
                if (vp->v_type != VDIR) {
                        error = ENOTDIR;
                        break;
                }

                /* must not be current dir */
                if (VOP_CMP(vp, cdir, ct)) {
                        error = EINVAL;
                        break;
                }

                /* see ufs_dirremove for why this is done, mount race */
                if (vn_vfswlock(vp)) {
                        error = EBUSY;
                        break;
                }
                vfslock = 1;
                if (vn_mountedvfs(vp) != NULL) {
                        error = EBUSY;
                        break;
                }
```

```
6333                 if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
6334                         error = cachefs_rmdir_connected(dvp, nm, cdir,
6335                             cr, vp);
6336                         if (CFS_TIMEOUT(fscp, error)) {
6337                                 cachefs_cd_release(fscp);
6338                                 held = 0;
6339                                 cachefs_cd_timedout(fscp);
6340                                 connected = 0;
6341                                 continue;
6342                         }
6343                 } else {
6344                         error = cachefs_rmdir_disconnected(dvp, nm, cdir,
6345                             cr, vp);
6346                         if (CFS_TIMEOUT(fscp, error)) {
6347                                 connected = 1;
6348                                 continue;
6349                         }
6350                 }
6351                 break;
6352         }

6354         if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_RMDIR)) {
6355                 ino64_t fileno = 0;
6356                 fid_t *fidp = NULL;
6357                 cnode_t *cp = NULL;
6358                 if (vp)
6359                         cp = VTOC(vp);

6361                 if (cp != NULL) {
6362                         fidp = &cp->c_metadata.md_cookie;
6363                         fileno = cp->c_id.cid_fileno;
6364                 }

6366                 cachefs_log_rmdir(cachep, error, fscp->fs_cfsvfsp,
6367                     fidp, fileno, crgetuid(cr));
6368         }

6370         if (held) {
6371                 cachefs_cd_release(fscp);
6372         }

6374         if (vfslock)
6375                 vn_vfsunlock(vp);

6377         if (vp)
6378                 VN_RELE(vp);

6380 #ifdef CFS_CD_DEBUG
6381         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
6382 #endif
6383 out:
6384 #ifdef CFSDEBUG
6385         CFS_DEBUG(CFSDEBUG_VOPS)
6386                 printf("cachefs_rmdir: EXIT error = %d\n", error);
6387 #endif

6389         return (error);
6390 }

6392 static int
6393 cachefs_rmdir_connected(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
6394     vnode_t *vp)
6395 {
6396         cnode_t *dcp = VTOC(dvp);
6397         cnode_t *cp = VTOC(vp);
```



```
6398         int error = 0;
6399         fscache_t *fscp = C_TO_FSCACHE(dcp);

6401         rw_enter(&dcp->c_rwlock, RW_WRITER);
6402         mutex_enter(&dcp->c_statelock);
6403         mutex_enter(&cp->c_statelock);

6405         if (dcp->c_backvp == NULL) {
6406                 error = cachefs_getbackvp(fscp, dcp);
6407                 if (error) {
6408                         goto out;
6409                 }
6410         }

6412         error = CFSOP_CHECK_COBJECT(fscp, dcp, 0, cr);
6413         if (error)
6414                 goto out;

6416         /* rmdir on the back fs */
6417         CFS_DPRINT_BACKFS_NFSV4(fscp,
6418             ("cachefs_rmdir (nfsv4): dcp %p, dbackvp %p, "
6419             "name %s\n", dcp, dcp->c_backvp, nm));
6420         error = VOP_RMDIR(dcp->c_backvp, nm, cdir, cr, NULL, 0);
6421         if (error)
6422                 goto out;

6424         /* if the dir is populated, remove the entry from it */
6425         if (CFS_ISFS_NONSHARED(fscp) &&
6426             (dcp->c_metadata.md_flags & MD_POPULATED)) {
6427                 error = cachefs_dir_rmentry(dcp, nm);
6428                 if (error) {
6429                         cachefs_nocache(dcp);
6430                         error = 0;
6431                 }
6432         }

6434         /*
6435          * *if* the (hard) link count goes to 0, then we set the CDESTROY
6436          * flag on the cnode. The cached object will then be destroyed
6437          * at inactive time where the chickens come home to roost :-)
6438          * The link cnt for directories is bumped down by 2 'cause the "."
6439          * entry has to be elided too ! The link cnt for the parent goes down
6440          * by 1 (because of "..").
6441          */
6442         cp->c_attr.va_nlink -= 2;
6443         dcp->c_attr.va_nlink--;
6444         if (cp->c_attr.va_nlink == 0) {
6445                 cp->c_flags |= CN_DESTROY;
6446         } else {
6447                 cp->c_flags |= CN_UPDATED;
6448         }
6449         dcp->c_flags |= CN_UPDATED;

6451         dnlc_purge_vp(vp);
6452         CFSOP_MODIFY_COBJECT(fscp, dcp, cr);

6454 out:
6455         mutex_exit(&cp->c_statelock);
6456         mutex_exit(&dcp->c_statelock);
6457         rw_exit(&dcp->c_rwlock);

6459         return (error);
6460 }

6462 static int
6463 /*ARGSUSED*/
```

```
6464 cachefs_rmdir_disconnected(vnode_t *dvp, char *nm, vnode_t *cdir,
6465     cred_t *cr, vnode_t *vp)
6466 {
6467         cnode_t *dcp = VTOC(dvp);
6468         cnode_t *cp = VTOC(vp);
6469         fscache_t *fscp = C_TO_FSCACHE(dcp);
6470         int error = 0;
6471         off_t commit = 0;
6472         timestruc_t current_time;

6474         if (CFS_ISFS_WRITE_AROUND(fscp))
6475                 return (ETIMEDOUT);

6477         rw_enter(&dcp->c_rwlock, RW_WRITER);
6478         mutex_enter(&dcp->c_statelock);
6479         mutex_enter(&cp->c_statelock);

6481         /* both directories must be populated */
6482         if (((dcp->c_metadata.md_flags & MD_POPULATED) == 0) ||
6483             ((cp->c_metadata.md_flags & MD_POPULATED) == 0)) {
6484                 error = ETIMEDOUT;
6485                 goto out;
6486         }

6488         /* if sticky bit set on the dir, more access checks to perform */
6489         if (error = cachefs_stickyrmchk(dcp, cp, cr)) {
6490                 goto out;
6491         }

6493         /* make sure dir is empty */
6494         if (cp->c_attr.va_nlink > 2) {
6495                 error = cachefs_dir_empty(cp);
6496                 if (error) {
6497                         if (error == ENOTDIR)
6498                                 error = ETIMEDOUT;
6499                         goto out;
6500                 }
6501                 cachefs_modified(cp);
6502         }
6503         cachefs_modified(dcp);

6505         /* log the operation */
6506         commit = cachefs_dlog_rmdir(fscp, dcp, nm, cp, cr);
6507         if (commit == 0) {
6508                 error = ENOSPC;
6509                 goto out;
6510         }

6512         /* remove name from parent dir */
6513         error = cachefs_dir_rmentry(dcp, nm);
6514         if (error == ENOTDIR) {
6515                 error = ETIMEDOUT;
6516                 goto out;
6517         }
6518         if (error)
6519                 goto out;

6521         gethrestime(&current_time);

6523         /* update deleted dir values */
6524         cp->c_attr.va_nlink -= 2;
6525         if (cp->c_attr.va_nlink == 0)
6526                 cp->c_flags |= CN_DESTROY;
6527         else {
6528                 cp->c_metadata.md_localctime = current_time;
6529                 cp->c_metadata.md_flags |= MD_LOCALCTIME;
```



```
6530                 cp->c_flags |= CN_UPDATED;
6531         }

6533         /* update parent values */
6534         dcp->c_metadata.md_localctime = current_time;
6535         dcp->c_metadata.md_localmtime = current_time;
6536         dcp->c_metadata.md_flags |= MD_LOCALCTIME | MD_LOCALMTIME;
6537         dcp->c_attr.va_nlink--;
6538         dcp->c_flags |= CN_UPDATED;

6540 out:
6541         mutex_exit(&cp->c_statelock);
6542         mutex_exit(&dcp->c_statelock);
6543         rw_exit(&dcp->c_rwlock);
6544         if (commit) {
6545                 /* commit the log entry */
6546                 if (cachefs_dlog_commit(fscp, commit, error)) {
6547                         /*EMPTY*/
6548                         /* XXX bob: fix on panic */
6549                 }
6550                 dnlc_purge_vp(vp);
6551         }
6552         return (error);
6553 }

6555 /*ARGSUSED*/
6556 static int
6557 cachefs_symlink(vnode_t *dvp, char *lnm, vattr_t *tva,
6558     char *tnm, cred_t *cr, caller_context_t *ct, int flags)
6559 {
6560         cnode_t *dcp = VTOC(dvp);
6561         fscache_t *fscp = C_TO_FSCACHE(dcp);
6562         cachefscache_t *cachep = fscp->fs_cache;
6563         int error = 0;
6564         int held = 0;
6565         int connected = 0;

6567 #ifdef CFSDEBUG
6568         CFS_DEBUG(CFSDEBUG_VOPS)
6569                 printf("cachefs_symlink: ENTER dvp %p lnm %s tnm %s\n",
6570                     (void *)dvp, lnm, tnm);
6571 #endif

6573         if (getzoneid() != GLOBAL_ZONEID) {
6574                 error = EPERM;
6575                 goto out;
6576         }

6578         if (fscp->fs_cache->c_flags & CACHE_NOCACHE)
6579                 ASSERT(dcp->c_flags & CN_NOCACHE);

6581         /*
6582          * Cachefs only provides pass-through support for NFSv4,
6583          * and all vnode operations are passed through to the
6584          * back file system. For NFSv4 pass-through to work, only
6585          * connected operation is supported, the cnode backvp must
6586          * exist, and cachefs optional (eg., disconnectable) flags
6587          * are turned off. Assert these conditions to ensure that
6588          * the backfilesystem is called for the symlink operation.
6589          */
6590         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
6591         CFS_BACKFS_NFSV4_ASSERT_CNODE(dcp);

6593         for (;;) {
6594                 /* get (or renew) access to the file system */
6595                 if (held) {
```

```
6596                        /* Won't loop with NFSv4 connected behavior */
6597                        ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
6598                        rw_exit(&dcp->c_rwlock);
6599                        cachefs_cd_release(fscp);
6600                        held = 0;
6601                }
6602                error = cachefs_cd_access(fscp, connected, 1);
6603                if (error)
6604                        break;
6605                rw_enter(&dcp->c_rwlock, RW_WRITER);
6606                held = 1;

6608                if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
6609                        error = cachefs_symlink_connected(dvp, lnm, tva,
6610                            tnm, cr);
6611                        if (CFS_TIMEOUT(fscp, error)) {
6612                                rw_exit(&dcp->c_rwlock);
6613                                cachefs_cd_release(fscp);
6614                                held = 0;
6615                                cachefs_cd_timedout(fscp);
6616                                connected = 0;
6617                                continue;
6618                        }
6619                } else {
6620                        error = cachefs_symlink_disconnected(dvp, lnm, tva,
6621                            tnm, cr);
6622                        if (CFS_TIMEOUT(fscp, error)) {
6623                                connected = 1;
6624                                continue;
6625                        }
6626                }
6627                break;
6628        }

6630        if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_SYMLINK))
6631                cachefs_log_symlink(cachep, error, fscp->fs_cfsvfsp,
6632                    &dcp->c_metadata.md_cookie, dcp->c_id.cid_fileno,
6633                    crgetuid(cr), (uint_t)strlen(tnm));

6635        if (held) {
6636                rw_exit(&dcp->c_rwlock);
6637                cachefs_cd_release(fscp);
6638        }

6640 #ifdef CFS_CD_DEBUG
6641        ASSERT((curthread->t_flag & T_CD_HELD) == 0);
6642 #endif
6643 out:
6644 #ifdef CFSDEBUG
6645        CFS_DEBUG(CFSDEBUG_VOPS)
6646                printf("cachefs_symlink: EXIT error = %d\n", error);
6647 #endif
6648        return (error);
6649 }

6651 static int
6652 cachefs_symlink_connected(vnode_t *dvp, char *lnm, vattr_t *tva,
6653     char *tnm, cred_t *cr)
6654 {
6655        cnode_t *dcp = VTOC(dvp);
6656        fscache_t *fscp = C_TO_FSCACHE(dcp);
6657        int error = 0;
6658        vnode_t *backvp = NULL;
6659        cnode_t *newcp = NULL;
6660        struct vattr va;
6661        struct fid cookie;
```



```
6662        cfs_cid_t cid;
6663        uint32_t valid_fid;

6665        mutex_enter(&dcp->c_statelock);

6667        if (dcp->c_backvp == NULL) {
6668                error = cachefs_getbackvp(fscp, dcp);
6669                if (error) {
6670                        cachefs_nocache(dcp);
6671                        mutex_exit(&dcp->c_statelock);
6672                        goto out;
6673                }
6674        }

6676        error = CFSOP_CHECK_COBJECT(fscp, dcp, 0, cr);
6677        if (error) {
6678                mutex_exit(&dcp->c_statelock);
6679                goto out;
6680        }
6681        CFS_DPRINT_BACKFS_NFSV4(fscp,
6682            ("cachefs_symlink (nfsv4): dcp %p, dbackvp %p, "
6683            "lnm %s, tnm %s\n", dcp, dcp->c_backvp, lnm, tnm));
6684        error = VOP_SYMLINK(dcp->c_backvp, lnm, tva, tnm, cr, NULL, 0);
6685        if (error) {
6686                mutex_exit(&dcp->c_statelock);
6687                goto out;
6688        }
6689        if ((dcp->c_filegrp->fg_flags & CFS_FG_WRITE) == 0 &&
6690            !CFS_ISFS_BACKFS_NFSV4(fscp)) {
6691                cachefs_nocache(dcp);
6692                mutex_exit(&dcp->c_statelock);
6693                goto out;
6694        }

6696        CFSOP_MODIFY_COBJECT(fscp, dcp, cr);

6698        /* lookup the symlink we just created and get its fid and attrs */
6699        (void) VOP_LOOKUP(dcp->c_backvp, lnm, &backvp, NULL, 0, NULL, cr,
6700            NULL, NULL, NULL);
6701        if (backvp == NULL) {
6702                if (CFS_ISFS_BACKFS_NFSV4(fscp) == 0)
6703                        cachefs_nocache(dcp);
6704                mutex_exit(&dcp->c_statelock);
6705                goto out;
6706        }

6708        valid_fid = (CFS_ISFS_BACKFS_NFSV4(fscp) ? FALSE : TRUE);
6709        error = cachefs_getcookie(backvp, &cookie, &va, cr, valid_fid);
6710        if (error) {
6711                ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
6712                error = 0;
6713                cachefs_nocache(dcp);
6714                mutex_exit(&dcp->c_statelock);
6715                goto out;
6716        }
6717        cid.cid_fileno = va.va_nodeid;
6718        cid.cid_flags = 0;

6720        /* if the dir is cached, add the symlink to it */
6721        if (CFS_ISFS_NONSHARED(fscp) &&
6722            (dcp->c_metadata.md_flags & MD_POPULATED)) {
6723                error = cachefs_dir_enter(dcp, lnm, &cookie, &cid, SM_ASYNC);
6724                if (error) {
6725                        cachefs_nocache(dcp);
6726                        error = 0;
6727                }
```

```
6728         }
6729         mutex_exit(&dcp->c_statelock);

6731         /* make the cnode for the sym link */
6732         error = cachefs_cnode_make(&cid, fscp, (valid_fid ? &cookie : NULL),
6733             &va, backvp, cr, 0, &newcp);
6734         if (error) {
6735                 ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
6736                 cachefs_nocache(dcp);
6737                 error = 0;
6738                 goto out;
6739         }

6741         /* try to cache the symlink contents */
6742         rw_enter(&newcp->c_rwlock, RW_WRITER);
6743         mutex_enter(&newcp->c_statelock);

6745         /*
6746          * try to cache the sym link, note that its a noop if NOCACHE
6747          * or NFSv4 is set
6748          */
6749         error = cachefs_stuffsymlink(newcp, tnm, (int)newcp->c_size);
6750         if (error) {
6751                 cachefs_nocache(newcp);
6752                 error = 0;
6753         }
6754         mutex_exit(&newcp->c_statelock);
6755         rw_exit(&newcp->c_rwlock);

6757 out:
6758         if (backvp)
6759                 VN_RELE(backvp);
6760         if (newcp)
6761                 VN_RELE(CTOV(newcp));
6762         return (error);
6763 }

6765 static int
6766 cachefs_symlink_disconnected(vnode_t *dvp, char *lnm, vattr_t *tva,
6767     char *tnm, cred_t *cr)
6768 {
6769         cnode_t *dcp = VTOC(dvp);
6770         fscache_t *fscp = C_TO_FSCACHE(dcp);
6771         int error;
6772         cnode_t *newcp = NULL;
6773         struct vattr va;
6774         timestruc_t current_time;
6775         off_t commit = 0;

6777         if (CFS_ISFS_WRITE_AROUND(fscp))
6778                 return (ETIMEDOUT);

6780         mutex_enter(&dcp->c_statelock);

6782         /* check permissions */
6783         if (error = cachefs_access_local(dcp, (VEXEC|VWRITE), cr)) {
6784                 mutex_exit(&dcp->c_statelock);
6785                 goto out;
6786         }

6788         /* the directory front file must be populated */
6789         if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
6790                 error = ETIMEDOUT;
6791                 mutex_exit(&dcp->c_statelock);
6792                 goto out;
6793         }
```



```
6795         /* make sure lnm does not already exist in the directory */
6796         error = cachefs_dir_look(dcp, lnm, NULL, NULL, NULL, NULL);
6797         if (error == ENOTDIR) {
6798                 error = ETIMEDOUT;
6799                 mutex_exit(&dcp->c_statelock);
6800                 goto out;
6801         }
6802         if (error != ENOENT) {
6803                 error = EEXIST;
6804                 mutex_exit(&dcp->c_statelock);
6805                 goto out;
6806         }

6808         /* make up a reasonable set of attributes */
6809         cachefs_attr_setup(tva, &va, dcp, cr);
6810         va.va_type = VLNK;
6811         va.va_mode |= S_IFLNK;
6812         va.va_size = strlen(tnm);

6814         mutex_exit(&dcp->c_statelock);

6816         /* create the cnode */
6817         error = cachefs_cnode_create(fscp, &va, 0, &newcp);
6818         if (error)
6819                 goto out;

6821         rw_enter(&newcp->c_rwlock, RW_WRITER);
6822         mutex_enter(&newcp->c_statelock);

6824         error = cachefs_dlog_cidmap(fscp);
6825         if (error) {
6826                 mutex_exit(&newcp->c_statelock);
6827                 rw_exit(&newcp->c_rwlock);
6828                 error = ENOSPC;
6829                 goto out;
6830         }

6832         cachefs_creategid(dcp, newcp, tva, cr);
6833         mutex_enter(&dcp->c_statelock);
6834         cachefs_createacl(dcp, newcp);
6835         mutex_exit(&dcp->c_statelock);
6836         gethrestime(&current_time);
6837         newcp->c_metadata.md_vattr.va_atime = current_time;
6838         newcp->c_metadata.md_localctime = current_time;
6839         newcp->c_metadata.md_localmtime = current_time;
6840         newcp->c_metadata.md_flags |= MD_MAPPING | MD_LOCALMTIME |
6841             MD_LOCALCTIME;
6842         newcp->c_flags |= CN_UPDATED;

6844         /* log the operation */
6845         commit = cachefs_dlog_symlink(fscp, dcp, newcp, lnm, tva, tnm, cr);
6846         if (commit == 0) {
6847                 mutex_exit(&newcp->c_statelock);
6848                 rw_exit(&newcp->c_rwlock);
6849                 error = ENOSPC;
6850                 goto out;
6851         }

6853         /* store the symlink contents */
6854         error = cachefs_stuffsymlink(newcp, tnm, (int)newcp->c_size);
6855         if (error) {
6856                 mutex_exit(&newcp->c_statelock);
6857                 rw_exit(&newcp->c_rwlock);
6858                 goto out;
6859         }
```

```
6860         if (cachefs_modified_alloc(newcp)) {
6861                 mutex_exit(&newcp->c_statelock);
6862                 rw_exit(&newcp->c_rwlock);
6863                 error = ENOSPC;
6864                 goto out;
6865         }

6867         /*
6868          * write the metadata now rather than waiting until
6869          * inactive so that if there's no space we can let
6870          * the caller know.
6871          */
6872         if (newcp->c_flags & CN_ALLOC_PENDING) {
6873                 if (newcp->c_filegrp->fg_flags & CFS_FG_ALLOC_ATTR) {
6874                         (void) filegrp_allocattr(newcp->c_filegrp);
6875                 }
6876                 error = filegrp_create_metadata(newcp->c_filegrp,
6877                     &newcp->c_metadata, &newcp->c_id);
6878                 if (error) {
6879                         mutex_exit(&newcp->c_statelock);
6880                         rw_exit(&newcp->c_rwlock);
6881                         goto out;
6882                 }
6883                 newcp->c_flags &= ~CN_ALLOC_PENDING;
6884         }
6885         error = filegrp_write_metadata(newcp->c_filegrp,
6886             &newcp->c_id, &newcp->c_metadata);
6887         if (error) {
6888                 mutex_exit(&newcp->c_statelock);
6889                 rw_exit(&newcp->c_rwlock);
6890                 goto out;
6891         }
6892         mutex_exit(&newcp->c_statelock);
6893         rw_exit(&newcp->c_rwlock);

6895         mutex_enter(&dcp->c_statelock);

6897         /* enter the new file in the directory */
6898         if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
6899                 error = ETIMEDOUT;
6900                 mutex_exit(&dcp->c_statelock);
6901                 goto out;
6902         }
6903         cachefs_modified(dcp);
6904         error = cachefs_dir_enter(dcp, lnm, &newcp->c_metadata.md_cookie,
6905             &newcp->c_id, SM_ASYNC);
6906         if (error) {
6907                 mutex_exit(&dcp->c_statelock);
6908                 goto out;
6909         }

6911         /* update parent dir times */
6912         dcp->c_metadata.md_localctime = current_time;
6913         dcp->c_metadata.md_localmtime = current_time;
6914         dcp->c_metadata.md_flags |= MD_LOCALMTIME | MD_LOCALCTIME;
6915         dcp->c_flags |= CN_UPDATED;
6916         mutex_exit(&dcp->c_statelock);

6918 out:
6919         if (commit) {
6920                 /* commit the log entry */
6921                 if (cachefs_dlog_commit(fscp, commit, error)) {
6922                         /*EMPTY*/
6923                         /* XXX bob: fix on panic */
6924                 }
6925         }
```



```
6927         if (error) {
6928                 if (newcp) {
6929                         mutex_enter(&newcp->c_statelock);
6930                         newcp->c_flags |= CN_DESTROY;
6931                         mutex_exit(&newcp->c_statelock);
6932                 }
6933         }
6934         if (newcp) {
6935                 VN_RELE(CTOV(newcp));
6936         }

6938         return (error);
6939 }

6941 /*ARGSUSED*/
6942 static int
6943 cachefs_readdir(vnode_t *vp, uio_t *uiop, cred_t *cr, int *eofp,
6944     caller_context_t *ct, int flags)
6945 {
6946         cnode_t *dcp = VTOC(vp);
6947         fscache_t *fscp = C_TO_FSCACHE(dcp);
6948         cachefscache_t *cachep = fscp->fs_cache;
6949         int error = 0;
6950         int held = 0;
6951         int connected = 0;

6953 #ifdef CFSDEBUG
6954         CFS_DEBUG(CFSDEBUG_VOPS)
6955                 printf("cachefs_readdir: ENTER vp %p\n", (void *)vp);
6956 #endif
6957         if (getzoneid() != GLOBAL_ZONEID) {
6958                 error = EPERM;
6959                 goto out;
6960         }

6962         /*
6963          * Cachefs only provides pass-through support for NFSv4,
6964          * and all vnode operations are passed through to the
6965          * back file system. For NFSv4 pass-through to work, only
6966          * connected operation is supported, the cnode backvp must
6967          * exist, and cachefs optional (eg., disconnectable) flags
6968          * are turned off. Assert these conditions to ensure that
6969          * the backfilesystem is called for the readdir operation.
6970          */
6971         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
6972         CFS_BACKFS_NFSV4_ASSERT_CNODE(dcp);

6974         for (;;) {
6975                 /* get (or renew) access to the file system */
6976                 if (held) {
6977                         /* Won't loop with NFSv4 connected behavior */
6978                         ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
6979                         rw_exit(&dcp->c_rwlock);
6980                         cachefs_cd_release(fscp);
6981                         held = 0;
6982                 }
6983                 error = cachefs_cd_access(fscp, connected, 0);
6984                 if (error)
6985                         break;
6986                 rw_enter(&dcp->c_rwlock, RW_READER);
6987                 held = 1;

6989                 /* quit if link count of zero (posix) */
6990                 if (dcp->c_attr.va_nlink == 0) {
6991                         if (eofp)
```

```
6992                                *eofp = 1;
6993                        error = 0;
6994                        break;
6995                }

6997                if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
6998                        error = cachefs_readdir_connected(vp, uiop, cr,
6999                            eofp);
7000                        if (CFS_TIMEOUT(fscp, error)) {
7001                                rw_exit(&dcp->c_rwlock);
7002                                cachefs_cd_release(fscp);
7003                                held = 0;
7004                                cachefs_cd_timedout(fscp);
7005                                connected = 0;
7006                                continue;
7007                        }
7008                } else {
7009                        error = cachefs_readdir_disconnected(vp, uiop, cr,
7010                            eofp);
7011                        if (CFS_TIMEOUT(fscp, error)) {
7012                                if (cachefs_cd_access_miss(fscp)) {
7013                                        error = cachefs_readdir_connected(vp,
7014                                            uiop, cr, eofp);
7015                                        if (!CFS_TIMEOUT(fscp, error))
7016                                                break;
7017                                        delay(5*hz);
7018                                        connected = 0;
7019                                        continue;
7020                                }
7021                                connected = 1;
7022                                continue;
7023                        }
7024                }
7025                break;
7026        }

7028        if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_READDIR))
7029                cachefs_log_readdir(cachep, error, fscp->fs_cfsvfsp,
7030                    &dcp->c_metadata.md_cookie, dcp->c_id.cid_fileno,
7031                    crgetuid(cr), uiop->uio_loffset, *eofp);

7033        if (held) {
7034                rw_exit(&dcp->c_rwlock);
7035                cachefs_cd_release(fscp);
7036        }

7038 #ifdef CFS_CD_DEBUG
7039        ASSERT((curthread->t_flag & T_CD_HELD) == 0);
7040 #endif
7041 out:
7042 #ifdef CFSDEBUG
7043        CFS_DEBUG(CFSDEBUG_VOPS)
7044                printf("cachefs_readdir: EXIT error = %d\n", error);
7045 #endif

7047        return (error);
7048 }

7050 static int
7051 cachefs_readdir_connected(vnode_t *vp, uio_t *uiop, cred_t *cr, int *eofp)
7052 {
7053        cnode_t *dcp = VTOC(vp);
7054        int error;
7055        fscache_t *fscp = C_TO_FSCACHE(dcp);
7056        struct cachefs_req *rp;
```



```
7058        mutex_enter(&dcp->c_statelock);

7060        /* check directory consistency */
7061        error = CFSOP_CHECK_COBJECT(fscp, dcp, 0, cr);
7062        if (error)
7063                goto out;
7064        dcp->c_usage++;

7066        /* if dir was modified, toss old contents */
7067        if (dcp->c_metadata.md_flags & MD_INVALREADDIR) {
7068                ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
7069                cachefs_inval_object(dcp);
7070        }

7072        error = 0;
7073        if (((dcp->c_metadata.md_flags & MD_POPULATED) == 0) &&
7074            ((dcp->c_flags & (CN_ASYNC_POPULATE | CN_NOCACHE)) == 0) &&
7075            !CFS_ISFS_BACKFS_NFSV4(fscp) &&
7076            (fscp->fs_cdconnected == CFS_CD_CONNECTED)) {

7078                if (cachefs_async_okay()) {

7080                        /*
7081                         * Set up asynchronous request to fill this
7082                         * directory.
7083                         */

7085                        dcp->c_flags |= CN_ASYNC_POPULATE;

7087                        rp = kmem_cache_alloc(cachefs_req_cache, KM_SLEEP);
7088                        rp->cfs_cmd = CFS_POPULATE;
7089                        rp->cfs_req_u.cu_populate.cpop_vp = vp;
7090                        rp->cfs_cr = cr;

7092                        crhold(cr);
7093                        VN_HOLD(vp);

7095                        cachefs_addqueue(rp, &fscp->fs_workq);
7096                } else {
7097                        error = cachefs_dir_fill(dcp, cr);
7098                        if (error != 0)
7099                                cachefs_nocache(dcp);
7100                }
7101        }

7103        /* if front file is populated */
7104        if (((dcp->c_flags & (CN_NOCACHE | CN_ASYNC_POPULATE)) == 0) &&
7105            !CFS_ISFS_BACKFS_NFSV4(fscp) &&
7106            (dcp->c_metadata.md_flags & MD_POPULATED)) {
7107                ASSERT(CFS_ISFS_BACKFS_NFSV4(fscp) == 0);
7108                error = cachefs_dir_read(dcp, uiop, eofp);
7109                if (error == 0)
7110                        fscp->fs_stats.st_hits++;
7111        }

7113        /* if front file could not be used */
7114        if ((error != 0) ||
7115            CFS_ISFS_BACKFS_NFSV4(fscp) ||
7116            (dcp->c_flags & (CN_NOCACHE | CN_ASYNC_POPULATE)) ||
7117            ((dcp->c_metadata.md_flags & MD_POPULATED) == 0)) {

7119                if (error && !(dcp->c_flags & CN_NOCACHE) &&
7120                    !CFS_ISFS_BACKFS_NFSV4(fscp))
7121                        cachefs_nocache(dcp);

7123                /* get the back vp */
```

```
7124                 if (dcp->c_backvp == NULL) {
7125                         error = cachefs_getbackvp(fscp, dcp);
7126                         if (error)
7127                                 goto out;
7128                 }
7129
7130                 if (fscp->fs_inum_size > 0) {
7131                         error = cachefs_readback_translate(dcp, uiop, cr, eofp);
7132                 } else {
7133                         /* do the dir read from the back fs */
7134                         (void) VOP_RWLOCK(dcp->c_backvp,
7135                             V_WRITELOCK_FALSE, NULL);
7136                         CFS_DPRINT_BACKFS_NFSV4(fscp,
7137                             ("cachefs_readdir (nfsv4): "
7138                             "dcp %p, dbackvp %p\n", dcp, dcp->c_backvp));
7139                         error = VOP_READDIR(dcp->c_backvp, uiop, cr, eofp,
7140                             NULL, 0);
7141                         VOP_RWUNLOCK(dcp->c_backvp, V_WRITELOCK_FALSE, NULL);
7142                 }
7143
7144                 if (error == 0)
7145                         fscp->fs_stats.st_misses++;
7146         }
7147
7148 out:
7149         mutex_exit(&dcp->c_statelock);
7150
7151         return (error);
7152 }
7153
7154 static int
7155 cachefs_readback_translate(cnode_t *cp, uio_t *uiop, cred_t *cr, int *eofp)
7156 {
7157         int error = 0;
7158         fscache_t *fscp = C_TO_FSCACHE(cp);
7159         caddr_t buffy = NULL;
7160         int buffysize = MAXBSIZE;
7161         caddr_t chrp, end;
7162         ino64_t newinum;
7163         struct dirent64 *de;
7164         uio_t uioin;
7165         iovec_t iov;
7166
7167         ASSERT(cp->c_backvp != NULL);
7168         ASSERT(fscp->fs_inum_size > 0);
7169
7170         if (uiop->uio_resid < buffysize)
7171                 buffysize = (int)uiop->uio_resid;
7172         buffy = cachefs_kmem_alloc(buffysize, KM_SLEEP);
7173
7174         iov.iov_base = buffy;
7175         iov.iov_len = buffysize;
7176         uioin.uio_iov = &iov;
7177         uioin.uio_iovcnt = 1;
7178         uioin.uio_segflg = UIO_SYSSPACE;
7179         uioin.uio_fmode = 0;
7180         uioin.uio_extflg = UIO_COPY_CACHED;
7181         uioin.uio_loffset = uiop->uio_loffset;
7182         uioin.uio_resid = buffysize;
7183
7184         (void) VOP_RWLOCK(cp->c_backvp, V_WRITELOCK_FALSE, NULL);
7185         error = VOP_READDIR(cp->c_backvp, &uioin, cr, eofp, NULL, 0);
7186         VOP_RWUNLOCK(cp->c_backvp, V_WRITELOCK_FALSE, NULL);
7187
7188         if (error != 0)
7189                 goto out;
```



```
7191         end = buffy + buffysize - uioin.uio_resid;
7192
7193         mutex_exit(&cp->c_statelock);
7194         mutex_enter(&fscp->fs_fslock);
7195
7196
7197         for (chrp = buffy; chrp < end; chrp += de->d_reclen) {
7198                 de = (dirent64_t *)chrp;
7199                 newinum = cachefs_inum_real2fake(fscp, de->d_ino);
7200                 if (newinum == 0)
7201                         newinum = cachefs_fileno_conflict(fscp, de->d_ino);
7202                 de->d_ino = newinum;
7203         }
7204         mutex_exit(&fscp->fs_fslock);
7205         mutex_enter(&cp->c_statelock);
7206
7207         error = uiomove(buffy, end - buffy, UIO_READ, uiop);
7208         uiop->uio_loffset = uioin.uio_loffset;
7209
7210 out:
7211
7212         if (buffy != NULL)
7213                 cachefs_kmem_free(buffy, buffysize);
7214
7215         return (error);
7216 }
7217
7218 static int
7219 /*ARGSUSED*/
7220 cachefs_readdir_disconnected(vnode_t *vp, uio_t *uiop, cred_t *cr,
7221     int *eofp)
7222 {
7223         cnode_t *dcp = VTOC(vp);
7224         int error;
7225
7226         mutex_enter(&dcp->c_statelock);
7227         if ((dcp->c_metadata.md_flags & MD_POPULATED) == 0) {
7228                 error = ETIMEDOUT;
7229         } else {
7230                 error = cachefs_dir_read(dcp, uiop, eofp);
7231                 if (error == ENOTDIR)
7232                         error = ETIMEDOUT;
7233         }
7234         mutex_exit(&dcp->c_statelock);
7235
7236         return (error);
7237 }
7238
7239 /*ARGSUSED*/
7240 static int
7241 cachefs_fid(struct vnode *vp, struct fid *fidp, caller_context_t *ct)
7242 {
7243         int error = 0;
7244         struct cnode *cp = VTOC(vp);
7245         fscache_t *fscp = C_TO_FSCACHE(cp);
7246
7247         /*
7248          * Cachefs only provides pass-through support for NFSv4,
7249          * and all vnode operations are passed through to the
7250          * back file system. For NFSv4 pass-through to work, only
7251          * connected operation is supported, the cnode backvp must
7252          * exist, and cachefs optional (eg., disconnectable) flags
7253          * are turned off. Assert these conditions, then bail
7254          * as  NFSv4 doesn't support VOP_FID.
7255          */
```

```
7256         CFS_BACKFS_NFSV4_ASSERT_FSCACHE(fscp);
7257         CFS_BACKFS_NFSV4_ASSERT_CNODE(cp);
7258         if (CFS_ISFS_BACKFS_NFSV4(fscp)) {
7259                 return (ENOTSUP);
7260         }

7262         mutex_enter(&cp->c_statelock);
7263         if (fidp->fid_len < cp->c_metadata.md_cookie.fid_len) {
7264                 fidp->fid_len = cp->c_metadata.md_cookie.fid_len;
7265                 error = ENOSPC;
7266         } else {
7267                 bcopy(cp->c_metadata.md_cookie.fid_data, fidp->fid_data,
7268                     cp->c_metadata.md_cookie.fid_len);
7269                 fidp->fid_len = cp->c_metadata.md_cookie.fid_len;
7270         }
7271         mutex_exit(&cp->c_statelock);
7272         return (error);
7273 }

7275 /* ARGSUSED2 */
7276 static int
7277 cachefs_rwlock(struct vnode *vp, int write_lock, caller_context_t *ctp)
7278 {
7279         cnode_t *cp = VTOC(vp);

7281         /*
7282          * XXX - This is ifdef'ed out for now. The problem -
7283          * getdents() acquires the read version of rwlock, then we come
7284          * into cachefs_readdir() and that wants to acquire the write version
7285          * of this lock (if its going to populate the directory). This is
7286          * a problem, this can be solved by introducing another lock in the
7287          * cnode.
7288          */
7289 /* XXX */
7290         if (vp->v_type != VREG)
7291                 return (-1);
7292         if (write_lock)
7293                 rw_enter(&cp->c_rwlock, RW_WRITER);
7294         else
7295                 rw_enter(&cp->c_rwlock, RW_READER);
7296         return (write_lock);
7297 }

7299 /* ARGSUSED */
7300 static void
7301 cachefs_rwunlock(struct vnode *vp, int write_lock, caller_context_t *ctp)
7302 {
7303         cnode_t *cp = VTOC(vp);
7304         if (vp->v_type != VREG)
7305                 return;
7306         rw_exit(&cp->c_rwlock);
7307 }

7309 /* ARGSUSED */
7310 static int
7311 cachefs_seek(struct vnode *vp, offset_t ooff, offset_t *noffp,
7312     caller_context_t *ct)
7313 {
7314         return (0);
7315 }

7317 static int cachefs_lostpage = 0;
7318 /*
7319  * Return all the pages from [off..off+len] in file
7320  */
7321 /*ARGSUSED*/
```



```
7322 static int
7323 cachefs_getpage(struct vnode *vp, offset_t off, size_t len,
7324         uint_t *protp, struct page *pl[], size_t plsz, struct seg *seg,
7325         caddr_t addr, enum seg_rw rw, cred_t *cr, caller_context_t *ct)
7326 {
7327         cnode_t *cp = VTOC(vp);
7328         int error;
7329         fscache_t *fscp = C_TO_FSCACHE(cp);
7330         cachefscache_t *cachep = fscp->fs_cache;
7331         int held = 0;
7332         int connected = 0;

7334 #ifdef CFSDEBUG
7335         u_offset_t offx = (u_offset_t)off;

7337         CFS_DEBUG(CFSDEBUG_VOPS)
7338                 printf("cachefs_getpage: ENTER vp %p off %lld len %lu rw %d\n",
7339                     (void *)vp, offx, len, rw);
7340 #endif
7341         if (getzoneid() != GLOBAL_ZONEID) {
7342                 error = EPERM;
7343                 goto out;
7344         }

7346         if (vp->v_flag & VNOMAP) {
7347                 error = ENOSYS;
7348                 goto out;
7349         }

7351         /* Call backfilesystem if NFSv4 */
7352         if (CFS_ISFS_BACKFS_NFSV4(fscp)) {
7353                 error = cachefs_getpage_backfs_nfsv4(vp, off, len, protp, pl,
7354                     plsz, seg, addr, rw, cr);
7355                 goto out;
7356         }

7358         /* XXX sam: make this do an async populate? */
7359         if (pl == NULL) {
7360                 error = 0;
7361                 goto out;
7362         }
7363         if (protp != NULL)
7364                 *protp = PROT_ALL;

7366         for (;;) {
7367                 /* get (or renew) access to the file system */
7368                 if (held) {
7369                         cachefs_cd_release(fscp);
7370                         held = 0;
7371                 }
7372                 error = cachefs_cd_access(fscp, connected, 0);
7373                 if (error)
7374                         break;
7375                 held = 1;

7377                 /*
7378                  * If we are getting called as a side effect of a
7379                  * cachefs_write()
7380                  * operation the local file size might not be extended yet.
7381                  * In this case we want to be able to return pages of zeroes.
7382                  */
7383                 if ((u_offset_t)off + len >
7384                     ((cp->c_size + PAGEOFFSET) & (offset_t)PAGEMASK)) {
7385                         if (seg != segkmap) {
7386                                 error = EFAULT;
7387                                 break;
```

```
7388                        }
7389                }
7390                error = pvn_getpages(cachefs_getapage, vp, (u_offset_t)off,
7391                    len, protp, pl, plsz, seg, addr, rw, cr);
  23                if (len <= PAGESIZE)
  24                        error = cachefs_getapage(vp, (u_offset_t)off, len,
  25                            protp, pl, plsz, seg, addr, rw, cr);
  26                else
  27                        error = pvn_getpages(cachefs_getapage, vp,
  28                            (u_offset_t)off, len, protp, pl, plsz, seg, addr,
  29                            rw, cr);
7392                if (error == 0)
7393                        break;

7395                if (((cp->c_flags & CN_NOCACHE) && (error == ENOSPC)) ||
7396                    error == EAGAIN) {
7397                        connected = 0;
7398                        continue;
7399                }
7400                if (fscp->fs_cdconnected == CFS_CD_CONNECTED) {
7401                        if (CFS_TIMEOUT(fscp, error)) {
7402                                cachefs_cd_release(fscp);
7403                                held = 0;
7404                                cachefs_cd_timedout(fscp);
7405                                connected = 0;
7406                                continue;
7407                        }
7408                } else {
7409                        if (CFS_TIMEOUT(fscp, error)) {
7410                                if (cachefs_cd_access_miss(fscp)) {
  49                                        if (len <= PAGESIZE)
  50                                                error = cachefs_getapage_back(
  51                                                    vp, (u_offset_t)off,
  52                                                    len, protp, pl,
  53                                                    plsz, seg, addr, rw, cr);
  54                                        else
7411                                        error = pvn_getpages(
7412                                            cachefs_getapage_back, vp,
7413                                            (u_offset_t)off, len, protp, pl,
  57                                                    (u_offset_t)off, len,
  58                                                    protp, pl,
7414                                            plsz, seg, addr, rw, cr);
7415                                        if (!CFS_TIMEOUT(fscp, error) &&
7416                                            (error != EAGAIN))
7417                                                break;
7418                                        delay(5*hz);
7419                                        connected = 0;
7420                                        continue;
7421                                }
7422                                connected = 1;
7423                                continue;
7424                        }
7425                }
7426                break;
7427        }

7429        if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_GETPAGE))
7430                cachefs_log_getpage(cachep, error, vp->v_vfsp,
7431                    &cp->c_metadata.md_cookie, cp->c_id.cid_fileno,
7432                    crgetuid(cr), off, len);

7434        if (held) {
7435                cachefs_cd_release(fscp);
7436        }

7438 out:
```



```
7439 #ifdef CFS_CD_DEBUG
7440         ASSERT((curthread->t_flag & T_CD_HELD) == 0);
7441 #endif
7442 #ifdef CFSDEBUG
7443         CFS_DEBUG(CFSDEBUG_VOPS)
7444                 printf("cachefs_getpage: EXIT vp %p error %d\n",
7445                     (void *)vp, error);
7446 #endif
7447         return (error);
7448 }
______unchanged_portion_omitted_

7490 /*
7491  * Called from pvn_getpages to get a particular page.
 136  * Called from pvn_getpages or cachefs_getpage to get a particular page.
7492  */
7493 /*ARGSUSED*/
7494 static int
7495 cachefs_getapage(struct vnode *vp, u_offset_t off, size_t len, uint_t *protp,
7496         struct page *pl[], size_t plsz, struct seg *seg, caddr_t addr,
7497         enum seg_rw rw, cred_t *cr)
7498 {
7499         cnode_t *cp = VTOC(vp);
7500         page_t **ppp, *pp = NULL;
7501         fscache_t *fscp = C_TO_FSCACHE(cp);
7502         cachefscache_t *cachep = fscp->fs_cache;
7503         int error = 0;
7504         struct page **ourpl;
7505         struct page *ourstackpl[17]; /* see ASSERT() below for 17 */
7506         int index = 0;
7507         int downgrade;
7508         int have_statelock = 0;
7509         u_offset_t popoff;
7510         size_t popsize = 0;

7512         /*LINTED*/
7513         ASSERT(((DEF_POP_SIZE / PAGESIZE) + 1) <= 17);

7515         if (fscp->fs_info.fi_popsize > DEF_POP_SIZE)
7516                 ourpl = cachefs_kmem_alloc(sizeof (struct page *) *
7517                     ((fscp->fs_info.fi_popsize / PAGESIZE) + 1), KM_SLEEP);
7518         else
7519                 ourpl = ourstackpl;

7521         ourpl[0] = NULL;
7522         off = off & (offset_t)PAGEMASK;
7523 again:
7524         /*
7525          * Look for the page
7526          */
7527         if (page_exists(vp, off) == 0) {
7528                 /*
7529                  * Need to do work to get the page.
7530                  * Grab our lock because we are going to
7531                  * modify the state of the cnode.
7532                  */
7533                 if (! have_statelock) {
7534                         mutex_enter(&cp->c_statelock);
7535                         have_statelock = 1;
7536                 }
7537                 /*
7538                  * If we're in NOCACHE mode, we will need a backvp
7539                  */
7540                 if (cp->c_flags & CN_NOCACHE) {
7541                         if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
7542                                 error = ETIMEDOUT;
```

```
7543                                    goto out;
7544                            }
7545                            if (cp->c_backvp == NULL) {
7546                                    error = cachefs_getbackvp(fscp, cp);
7547                                    if (error)
7548                                            goto out;
7549                            }
7550                            error = VOP_GETPAGE(cp->c_backvp, off,
7551                                PAGESIZE, protp, ourpl, PAGESIZE, seg,
7552                                addr, S_READ, cr, NULL);
7553                            /*
7554                             * backfs returns EFAULT when we are trying for a
7555                             * page beyond EOF but cachefs has the knowledge that
7556                             * it is not beyond EOF be cause cp->c_size is
7557                             * greater then the offset requested.
7558                             */
7559                            if (error == EFAULT) {
7560                                    error = 0;
7561                                    pp = page_create_va(vp, off, PAGESIZE,
7562                                        PG_EXCL | PG_WAIT, seg, addr);
7563                                    if (pp == NULL)
7564                                            goto again;
7565                                    pagezero(pp, 0, PAGESIZE);
7566                                    pvn_plist_init(pp, pl, plsz, off, PAGESIZE, rw);
7567                                    goto out;
7568                            }
7569                            if (error)
7570                                    goto out;
7571                            goto getpages;
7572                    }
7573                    /*
7574                     * We need a front file. If we can't get it,
7575                     * put the cnode in NOCACHE mode and try again.
7576                     */
7577                    if (cp->c_frontvp == NULL) {
7578                            error = cachefs_getfrontfile(cp);
7579                            if (error) {
7580                                    cachefs_nocache(cp);
7581                                    error = EAGAIN;
7582                                    goto out;
7583                            }
7584                    }
7585                    /*
7586                     * Check if the front file needs population.
7587                     * If population is necessary, make sure we have a
7588                     * backvp as well. We will get the page from the backvp.
7589                     * bug 4152459-
7590                     * But if the file system is in disconnected mode
7591                     * and the file is a local file then do not check the
7592                     * allocmap.
7593                     */
7594                    if (((fscp->fs_cdconnected == CFS_CD_CONNECTED) ||
7595                        ((cp->c_metadata.md_flags & MD_LOCALFILENO) == 0)) &&
7596                        (cachefs_check_allocmap(cp, off) == 0)) {
7597                            if (fscp->fs_cdconnected != CFS_CD_CONNECTED) {
7598                                    error = ETIMEDOUT;
7599                                    goto out;
7600                            }
7601                            if (cp->c_backvp == NULL) {
7602                                    error = cachefs_getbackvp(fscp, cp);
7603                                    if (error)
7604                                            goto out;
7605                            }
7606                            if (cp->c_filegrp->fg_flags & CFS_FG_WRITE) {
7607                                    cachefs_cluster_allocmap(off, &popoff,
7608                                        &popsize,
```



```
7609                                        fscp->fs_info.fi_popsize, cp);
7610                                if (popsize != 0) {
7611                                        error = cachefs_populate(cp,
7612                                            popoff, popsize,
7613                                            cp->c_frontvp, cp->c_backvp,
7614                                            cp->c_size, cr);
7615                                        if (error) {
7616                                                cachefs_nocache(cp);
7617                                                error = EAGAIN;
7618                                                goto out;
7619                                        } else {
7620                                                cp->c_flags |=
7621                                                    CN_UPDATED |
7622                                                    CN_NEED_FRONT_SYNC |
7623                                                    CN_POPULATION_PENDING;
7624                                        }
7625                                        popsize = popsize - (off - popoff);
7626                                } else {
7627                                        popsize = PAGESIZE;
7628                                }
7629                        }
7630                        /* else XXX assert CN_NOCACHE? */
7631                        error = VOP_GETPAGE(cp->c_backvp, (offset_t)off,
7632                            PAGESIZE, protp, ourpl, popsize,
7633                            seg, addr, S_READ, cr, NULL);
7634                        if (error)
7635                                goto out;
7636                        fscp->fs_stats.st_misses++;
7637                } else {
7638                        if (cp->c_flags & CN_POPULATION_PENDING) {
7639                                error = VOP_FSYNC(cp->c_frontvp, FSYNC, cr,
7640                                    NULL);
7641                                cp->c_flags &= ~CN_POPULATION_PENDING;
7642                                if (error) {
7643                                        cachefs_nocache(cp);
7644                                        error = EAGAIN;
7645                                        goto out;
7646                                }
7647                        }
7648                        /*
7649                         * File was populated so we get the page from the
7650                         * frontvp
7651                         */
7652                        error = VOP_GETPAGE(cp->c_frontvp, (offset_t)off,
7653                            PAGESIZE, protp, ourpl, PAGESIZE, seg, addr,
7654                            rw, cr, NULL);
7655                        if (CACHEFS_LOG_LOGGING(cachep, CACHEFS_LOG_GPFRONT))
7656                                cachefs_log_gpfront(cachep, error,
7657                                    fscp->fs_cfsvfsp,
7658                                    &cp->c_metadata.md_cookie, cp->c_fileno,
7659                                    crgetuid(cr), off, PAGESIZE);
7660                        if (error) {
7661                                cachefs_nocache(cp);
7662                                error = EAGAIN;
7663                                goto out;
7664                        }
7665                        fscp->fs_stats.st_hits++;
7666                }
7667 getpages:
7668                ASSERT(have_statelock);
7669                if (have_statelock) {
7670                        mutex_exit(&cp->c_statelock);
7671                        have_statelock = 0;
7672                }
7673                downgrade = 0;
7674                for (ppp = ourpl; *ppp; ppp++) {
```

```
7675                        if ((*ppp)->p_offset < off) {
7676                                index++;
7677                                page_unlock(*ppp);
7678                                continue;
7679                        }
7680                        if (PAGE_SHARED(*ppp)) {
7681                                if (page_tryupgrade(*ppp) == 0) {
7682                                        for (ppp = &ourpl[index]; *ppp; ppp++)
7683                                                page_unlock(*ppp);
7684                                        error = EAGAIN;
7685                                        goto out;
7686                                }
7687                                downgrade = 1;
7688                        }
7689                        ASSERT(PAGE_EXCL(*ppp));
7690                        (void) hat_pageunload((*ppp), HAT_FORCE_PGUNLOAD);
7691                        page_rename(*ppp, vp, (*ppp)->p_offset);
7692                }
7693                pl[0] = ourpl[index];
7694                pl[1] = NULL;
7695                if (downgrade) {
7696                        page_downgrade(ourpl[index]);
7697                }
7698                /* Unlock the rest of the pages from the cluster */
7699                for (ppp = &ourpl[index+1]; *ppp; ppp++)
7700                        page_unlock(*ppp);
7701        } else {
7702                ASSERT(! have_statelock);
7703                if (have_statelock) {
7704                        mutex_exit(&cp->c_statelock);
7705                        have_statelock = 0;
7706                }
7707                /* XXX SE_SHARED probably isn't what we *always* want */
7708                if ((pp = page_lookup(vp, off, SE_SHARED)) == NULL) {
7709                        cachefs_lostpage++;
7710                        goto again;
7711                }
7712                pl[0] = pp;
7713                pl[1] = NULL;
7714                /* XXX increment st_hits?  i don't think so, but... */
7715        }

7717 out:
7718        if (have_statelock) {
7719                mutex_exit(&cp->c_statelock);
7720                have_statelock = 0;
7721        }
7722        if (fscp->fs_info.fi_popsize > DEF_POP_SIZE)
7723                cachefs_kmem_free(ourpl, sizeof (struct page *) *
7724                    ((fscp->fs_info.fi_popsize / PAGESIZE) + 1));
7725        return (error);
7726 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   63365 Thu Jan  8 09:14:34 2015
new/usr/src/uts/common/fs/hsfs/hsfs_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

/*
 * Vnode operations for the High Sierra filesystem
 */

#include <sys/types.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/resource.h>
#include <sys/signal.h>
#include <sys/cred.h>
#include <sys/user.h>
#include <sys/buf.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/stat.h>
#include <sys/vnode.h>
#include <sys/mode.h>
#include <sys/proc.h>
#include <sys/disp.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/kmem.h>
#include <sys/uio.h>
#include <sys/conf.h>
#include <sys/errno.h>
#include <sys/mman.h>
#include <sys/pathname.h>
#include <sys/debug.h>
#include <sys/vmsystm.h>
```



```
#include <sys/cmn_err.h>
#include <sys/fbuf.h>
#include <sys/dirent.h>
#include <sys/errno.h>
#include <sys/dkio.h>
#include <sys/cmn_err.h>
#include <sys/atomic.h>

#include <vm/hat.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/as.h>
#include <vm/seg.h>
#include <vm/seg_map.h>
#include <vm/seg_kmem.h>
#include <vm/seg_vn.h>
#include <vm/rm.h>
#include <vm/page.h>
#include <sys/swap.h>
#include <sys/avl.h>
#include <sys/sunldi.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sdt.h>

/*
 * For struct modlinkage
 */
#include <sys/modctl.h>

#include <sys/fs/hsfs_spec.h>
#include <sys/fs/hsfs_node.h>
#include <sys/fs/hsfs_impl.h>
#include <sys/fs/hsfs_susp.h>
#include <sys/fs/hsfs_rrip.h>

#include <fs/fs_subr.h>

/* # of contiguous requests to detect sequential access pattern */
static int seq_contig_requests = 2;

/*
 * This is the max number os taskq threads that will be created
 * if required. Since we are using a Dynamic TaskQ by default only
 * one thread is created initially.
 *
 * NOTE: In the usual hsfs use case this per fs instance number
 * of taskq threads should not place any undue load on a system.
 * Even on an unusual system with say 100 CDROM drives, 800 threads
 * will not be created unless all the drives are loaded and all
 * of them are saturated with I/O at the same time! If there is at
 * all a complaint of system load due to such an unusual case it
 * should be easy enough to change to one per-machine Dynamic TaskQ
 * for all hsfs mounts with a nthreads of say 32.
 */
static int hsfs_taskq_nthreads = 8;     /* # of taskq threads per fs */

/* Min count of adjacent bufs that will avoid buf coalescing */
static int hsched_coalesce_min = 2;

/*
 * Kmem caches for heavily used small allocations. Using these kmem
 * caches provides a factor of 3 reduction in system time and greatly
 * aids overall throughput esp. on SPARC.
 */
struct kmem_cache *hio_cache;
```

```
128 struct kmem_cache *hio_info_cache;

130 /*
131  * This tunable allows us to ignore inode numbers from rrip-1.12.
132  * In this case, we fall back to our default inode algorithm.
133  */
134 extern int use_rrip_inodes;

136 /*
137  * Free behind logic from UFS to tame our thirst for
138  * the page cache.
139  * See usr/src/uts/common/fs/ufs/ufs_vnops.c for more
140  * explanation.
141  */
142 static int      freebehind = 1;
143 static int      smallfile = 0;
144 static int      cache_read_ahead = 0;
145 static u_offset_t smallfile64 = 32 * 1024;
146 #define SMALLFILE1_D 1000
147 #define SMALLFILE2_D 10
148 static u_offset_t smallfile1 = 32 * 1024;
149 static u_offset_t smallfile2 = 32 * 1024;
150 static clock_t smallfile_update = 0; /* when to recompute */
151 static uint_t smallfile1_d = SMALLFILE1_D;
152 static uint_t smallfile2_d = SMALLFILE2_D;

154 static int hsched_deadline_compare(const void *x1, const void *x2);
155 static int hsched_offset_compare(const void *x1, const void *x2);
156 static void hsched_enqueue_io(struct hsfs *fsp, struct hio *hsio, int ra);
157 int hsched_invoke_strategy(struct hsfs *fsp);

159 /* ARGSUSED */
160 static int
161 hsfs_fsync(vnode_t *cp,
162         int syncflag,
163         cred_t *cred,
164         caller_context_t *ct)
165 {
166         return (0);
167 }


170 /*ARGSUSED*/
171 static int
172 hsfs_read(struct vnode *vp,
173         struct uio *uiop,
174         int ioflag,
175         struct cred *cred,
176         struct caller_context *ct)
177 {
178         caddr_t base;
179         offset_t diff;
180         int error;
181         struct hsnode *hp;
182         uint_t filesize;
183         int dofree;

185         hp = VTOH(vp);
186         /*
187          * if vp is of type VDIR, make sure dirent
188          * is filled up with all info (because of ptbl)
189          */
190         if (vp->v_type == VDIR) {
191                 if (hp->hs_dirent.ext_size == 0)
192                         hs_filldirent(vp, &hp->hs_dirent);
193         }
```



```
194         filesize = hp->hs_dirent.ext_size;

196         /* Sanity checks. */
197         if (uiop->uio_resid == 0 ||             /* No data wanted. */
198             uiop->uio_loffset > HS_MAXFILEOFF ||        /* Offset too big. */
199             uiop->uio_loffset >= filesize)      /* Past EOF. */
200                 return (0);

202         do {
203                 /*
204                  * We want to ask for only the "right" amount of data.
205                  * In this case that means:-
206                  *
207                  * We can't get data from beyond our EOF. If asked,
208                  * we will give a short read.
209                  *
210                  * segmap_getmapflt returns buffers of MAXBSIZE bytes.
211                  * These buffers are always MAXBSIZE aligned.
212                  * If our starting offset is not MAXBSIZE aligned,
213                  * we can only ask for less than MAXBSIZE bytes.
214                  *
215                  * If our requested offset and length are such that
216                  * they belong in different MAXBSIZE aligned slots
217                  * then we'll be making more than one call on
218                  * segmap_getmapflt.
219                  *
220                  * This diagram shows the variables we use and their
221                  * relationships.
222                  *
223                  * |<-----MAXBSIZE----->|
224                  * +-------------------------...+
225                  * |.....mapon->|<--n-->|....*...|EOF
226                  * +-------------------------...+
227                  * uio_loffset->|
228                  * uio_resid....|<---------->|
229                  * diff.........|<-------------->|
230                  *
231                  * So, in this case our offset is not aligned
232                  * and our request takes us outside of the
233                  * MAXBSIZE window. We will break this up into
234                  * two segmap_getmapflt calls.
235                  */
236                 size_t nbytes;
237                 offset_t mapon;
238                 size_t n;
239                 uint_t flags;

241                 mapon = uiop->uio_loffset & MAXBOFFSET;
242                 diff = filesize - uiop->uio_loffset;
243                 nbytes = (size_t)MIN(MAXBSIZE - mapon, uiop->uio_resid);
244                 n = MIN(diff, nbytes);
245                 if (n <= 0) {
246                         /* EOF or request satisfied. */
247                         return (0);
248                 }

250                 /*
251                  * Freebehind computation taken from:
252                  * usr/src/uts/common/fs/ufs/ufs_vnops.c
253                  */
254                 if (drv_hztousec(ddi_get_lbolt()) >= smallfile_update) {
255                         uint64_t percpufreeb;
256                         if (smallfile1_d == 0) smallfile1_d = SMALLFILE1_D;
257                         if (smallfile2_d == 0) smallfile2_d = SMALLFILE2_D;
258                         percpufreeb = ptob((uint64_t)freemem) / ncpus_online;
259                         smallfile1 = percpufreeb / smallfile1_d;
```

```
260                         smallfile2 = percpufreeb / smallfile2_d;
261                         smallfile1 = MAX(smallfile1, smallfile);
262                         smallfile1 = MAX(smallfile1, smallfile64);
263                         smallfile2 = MAX(smallfile1, smallfile2);
264                         smallfile_update = drv_hztousec(ddi_get_lbolt())
265                             + 1000000;
266                 }
267
268                 dofree = freebehind &&
269                     hp->hs_prev_offset == uiop->uio_loffset &&
270                     hp->hs_ra_bytes > 0;
271
272                 base = segmap_getmapflt(segkmap, vp,
273                     (u_offset_t)uiop->uio_loffset, n, 1, S_READ);
274
275                 error = uiomove(base + mapon, n, UIO_READ, uiop);
276
277                 if (error == 0) {
278                         /*
279                          * if read a whole block, or read to eof,
280                          *  won't need this buffer again soon.
281                          */
282                         if (n + mapon == MAXBSIZE ||
283                             uiop->uio_loffset == filesize)
284                                 flags = SM_DONTNEED;
285                         else
286                                 flags = 0;
287
288                         if (dofree) {
289                                 flags = SM_FREE | SM_ASYNC;
290                                 if ((cache_read_ahead == 0) &&
291                                     uiop->uio_loffset > smallfile2)
292                                         flags |=  SM_DONTNEED;
293                         }
294
295                         error = segmap_release(segkmap, base, flags);
296                 } else
297                         (void) segmap_release(segkmap, base, 0);
298         } while (error == 0 && uiop->uio_resid > 0);
299
300         return (error);
301 }
302
303 /*ARGSUSED2*/
304 static int
305 hsfs_getattr(
306         struct vnode *vp,
307         struct vattr *vap,
308         int flags,
309         struct cred *cred,
310         caller_context_t *ct)
311 {
312         struct hsnode *hp;
313         struct vfs *vfsp;
314         struct hsfs *fsp;
315
316         hp = VTOH(vp);
317         fsp = VFS_TO_HSFS(vp->v_vfsp);
318         vfsp = vp->v_vfsp;
319
320         if ((hp->hs_dirent.ext_size == 0) && (vp->v_type == VDIR)) {
321                 hs_filldirent(vp, &hp->hs_dirent);
322         }
323         vap->va_type = IFTOVT(hp->hs_dirent.mode);
324         vap->va_mode = hp->hs_dirent.mode;
325         vap->va_uid = hp->hs_dirent.uid;
```



```
326         vap->va_gid = hp->hs_dirent.gid;
327
328         vap->va_fsid = vfsp->vfs_dev;
329         vap->va_nodeid = (ino64_t)hp->hs_nodeid;
330         vap->va_nlink = hp->hs_dirent.nlink;
331         vap->va_size =  (offset_t)hp->hs_dirent.ext_size;
332
333         vap->va_atime.tv_sec = hp->hs_dirent.adate.tv_sec;
334         vap->va_atime.tv_nsec = hp->hs_dirent.adate.tv_usec*1000;
335         vap->va_mtime.tv_sec = hp->hs_dirent.mdate.tv_sec;
336         vap->va_mtime.tv_nsec = hp->hs_dirent.mdate.tv_usec*1000;
337         vap->va_ctime.tv_sec = hp->hs_dirent.cdate.tv_sec;
338         vap->va_ctime.tv_nsec = hp->hs_dirent.cdate.tv_usec*1000;
339         if (vp->v_type == VCHR || vp->v_type == VBLK)
340                 vap->va_rdev = hp->hs_dirent.r_dev;
341         else
342                 vap->va_rdev = 0;
343         vap->va_blksize = vfsp->vfs_bsize;
344         /* no. of blocks = no. of data blocks + no. of xar blocks */
345         vap->va_nblocks = (fsblkcnt64_t)howmany(vap->va_size + (u_longlong_t)
346             (hp->hs_dirent.xar_len << fsp->hsfs_vol.lbn_shift), DEV_BSIZE);
347         vap->va_seq = hp->hs_seq;
348         return (0);
349 }
350
351 /*ARGSUSED*/
352 static int
353 hsfs_readlink(struct vnode *vp,
354         struct uio *uiop,
355         struct cred *cred,
356         caller_context_t *ct)
357 {
358         struct hsnode *hp;
359
360         if (vp->v_type != VLNK)
361                 return (EINVAL);
362
363         hp = VTOH(vp);
364
365         if (hp->hs_dirent.sym_link == (char *)NULL)
366                 return (ENOENT);
367
368         return (uiomove(hp->hs_dirent.sym_link,
369             (size_t)MIN(hp->hs_dirent.ext_size,
370             uiop->uio_resid), UIO_READ, uiop));
371 }
372
373 /*ARGSUSED*/
374 static void
375 hsfs_inactive(struct vnode *vp,
376         struct cred *cred,
377         caller_context_t *ct)
378 {
379         struct hsnode *hp;
380         struct hsfs *fsp;
381
382         int nopage;
383
384         hp = VTOH(vp);
385         fsp = VFS_TO_HSFS(vp->v_vfsp);
386         /*
387          * Note: acquiring and holding v_lock for quite a while
388          * here serializes on the vnode; this is unfortunate, but
389          * likely not to overly impact performance, as the underlying
390          * device (CDROM drive) is quite slow.
391          */
```

```
392         rw_enter(&fsp->hsfs_hash_lock, RW_WRITER);
393         mutex_enter(&hp->hs_contents_lock);
394         mutex_enter(&vp->v_lock);

396         if (vp->v_count < 1) {
397                 panic("hsfs_inactive: v_count < 1");
398                 /*NOTREACHED*/
399         }

401         if (vp->v_count > 1 || (hp->hs_flags & HREF) == 0) {
402                 vp->v_count--;  /* release hold from vn_rele */
403                 mutex_exit(&vp->v_lock);
404                 mutex_exit(&hp->hs_contents_lock);
405                 rw_exit(&fsp->hsfs_hash_lock);
406                 return;
407         }
408         vp->v_count--;  /* release hold from vn_rele */
409         if (vp->v_count == 0) {
410                 /*
411                  * Free the hsnode.
412                  * If there are no pages associated with the
413                  * hsnode, give it back to the kmem_cache,
414                  * else put at the end of this file system's
415                  * internal free list.
416                  */
417                 nopage = !vn_has_cached_data(vp);
418                 hp->hs_flags = 0;
419                 /*
420                  * exit these locks now, since hs_freenode may
421                  * kmem_free the hsnode and embedded vnode
422                  */
423                 mutex_exit(&vp->v_lock);
424                 mutex_exit(&hp->hs_contents_lock);
425                 hs_freenode(vp, fsp, nopage);
426         } else {
427                 mutex_exit(&vp->v_lock);
428                 mutex_exit(&hp->hs_contents_lock);
429         }
430         rw_exit(&fsp->hsfs_hash_lock);
431 }


434 /*ARGSUSED*/
435 static int
436 hsfs_lookup(
437         struct vnode *dvp,
438         char *nm,
439         struct vnode **vpp,
440         struct pathname *pnp,
441         int flags,
442         struct vnode *rdir,
443         struct cred *cred,
444         caller_context_t *ct,
445         int *direntflags,
446         pathname_t *realpnp)
447 {
448         int error;
449         int namelen = (int)strlen(nm);

451         if (*nm == '\0') {
452                 VN_HOLD(dvp);
453                 *vpp = dvp;
454                 return (0);
455         }

457         /*
```



```
458          * If we're looking for ourself, life is simple.
459          */
460         if (namelen == 1 && *nm == '.') {
461                 if (error = hs_access(dvp, (mode_t)VEXEC, cred))
462                         return (error);
463                 VN_HOLD(dvp);
464                 *vpp = dvp;
465                 return (0);
466         }

468         return (hs_dirlook(dvp, nm, namelen, vpp, cred));
469 }


472 /*ARGSUSED*/
473 static int
474 hsfs_readdir(
475         struct vnode            *vp,
476         struct uio              *uiop,
477         struct cred             *cred,
478         int                     *eofp,
479         caller_context_t        *ct,
480         int                     flags)
481 {
482         struct hsnode   *dhp;
483         struct hsfs     *fsp;
484         struct hs_direntry hd;
485         struct dirent64 *nd;
486         int             error;
487         uint_t          offset;         /* real offset in directory */
488         uint_t          dirsiz;         /* real size of directory */
489         uchar_t         *blkp;
490         int             hdlen;          /* length of hs directory entry */
491         long            ndlen;          /* length of dirent entry */
492         int             bytes_wanted;
493         size_t          bufsize;        /* size of dirent buffer */
494         char            *outbuf;        /* ptr to dirent buffer */
495         char            *dname;
496         int             dnamelen;
497         size_t          dname_size;
498         struct fbuf     *fbp;
499         uint_t          last_offset;    /* last index into current dir block */
500         ino64_t         dirino; /* temporary storage before storing in dirent */
501         off_t           diroff;

503         dhp = VTOH(vp);
504         fsp = VFS_TO_HSFS(vp->v_vfsp);
505         if (dhp->hs_dirent.ext_size == 0)
506                 hs_filldirent(vp, &dhp->hs_dirent);
507         dirsiz = dhp->hs_dirent.ext_size;
508         if (uiop->uio_loffset >= dirsiz) {      /* at or beyond EOF */
509                 if (eofp)
510                         *eofp = 1;
511                 return (0);
512         }
513         ASSERT(uiop->uio_loffset <= HS_MAXFILEOFF);
514         offset = uiop->uio_loffset;

516         dname_size = fsp->hsfs_namemax + 1;     /* 1 for the ending NUL */
517         dname = kmem_alloc(dname_size, KM_SLEEP);
518         bufsize = uiop->uio_resid + sizeof (struct dirent64);

520         outbuf = kmem_alloc(bufsize, KM_SLEEP);
521         nd = (struct dirent64 *)outbuf;

523         while (offset < dirsiz) {
```

```
524                 bytes_wanted = MIN(MAXBSIZE, dirsiz - (offset & MAXBMASK));
525
526                 error = fbread(vp, (offset_t)(offset & MAXBMASK),
527                     (unsigned int)bytes_wanted, S_READ, &fbp);
528                 if (error)
529                         goto done;
530
531                 blkp = (uchar_t *)fbp->fb_addr;
532                 last_offset = (offset & MAXBMASK) + fbp->fb_count;
533
534 #define rel_offset(offset) ((offset) & MAXBOFFSET)      /* index into blkp */
535
536                 while (offset < last_offset) {
537                         /*
538                          * Very similar validation code is found in
539                          * process_dirblock(), hsfs_node.c.
540                          * For an explanation, see there.
541                          * It may make sense for the future to
542                          * "consolidate" the code in hs_parsedir(),
543                          * process_dirblock() and hsfs_readdir() into
544                          * a single utility function.
545                          */
546                         hdlen = (int)((uchar_t)
547                             HDE_DIR_LEN(&blkp[rel_offset(offset)]));
548                         if (hdlen < HDE_ROOT_DIR_REC_SIZE ||
549                             offset + hdlen > last_offset) {
550                                 /*
551                                  * advance to next sector boundary
552                                  */
553                                 offset = roundup(offset + 1, HS_SECTOR_SIZE);
554                                 if (hdlen)
555                                         hs_log_bogus_disk_warning(fsp,
556                                             HSFS_ERR_TRAILING_JUNK, 0);
557
558                                 continue;
559                         }
560
561                         bzero(&hd, sizeof (hd));
562
563                         /*
564                          * Just ignore invalid directory entries.
565                          * XXX - maybe hs_parsedir() will detect EXISTENCE bit
566                          */
567                         if (!hs_parsedir(fsp, &blkp[rel_offset(offset)],
568                             &hd, dname, &dnamelen, last_offset - offset)) {
569                                 /*
570                                  * Determine if there is enough room
571                                  */
572                                 ndlen = (long)DIRENT64_RECLEN((dnamelen));
573
574                                 if ((ndlen + ((char *)nd - outbuf)) >
575                                     uiop->uio_resid) {
576                                         fbrelse(fbp, S_READ);
577                                         goto done; /* output buffer full */
578                                 }
579
580                                 diroff = offset + hdlen;
581                                 /*
582                                  * If the media carries rrip-v1.12 or newer,
583                                  * and we trust the inodes from the rrip data
584                                  * (use_rrip_inodes != 0), use that data. If the
585                                  * media has been created by a recent mkisofs
586                                  * version, we may trust all numbers in the
587                                  * starting extent number; otherwise, we cannot
588                                  * do this for zero sized files and symlinks,
589                                  * because if we did we'd end up mapping all of
```



```
590                                  * them to the same node. We use HS_DUMMY_INO
591                                  * in this case and make sure that we will not
592                                  * map all files to the same meta data.
593                                  */
594                                 if (hd.inode != 0 && use_rrip_inodes) {
595                                         dirino = hd.inode;
596                                 } else if ((hd.ext_size == 0 ||
597                                     hd.sym_link != (char *)NULL) &&
598                                     (fsp->hsfs_flags & HSFSMNT_INODE) == 0) {
599                                         dirino = HS_DUMMY_INO;
600                                 } else {
601                                         dirino = hd.ext_lbn;
602                                 }
603
604                                 /* strncpy(9f) will zero uninitialized bytes */
605
606                                 ASSERT(strlen(dname) + 1 <=
607                                     DIRENT64_NAMELEN(ndlen));
608                                 (void) strncpy(nd->d_name, dname,
609                                     DIRENT64_NAMELEN(ndlen));
610                                 nd->d_reclen = (ushort_t)ndlen;
611                                 nd->d_off = (offset_t)diroff;
612                                 nd->d_ino = dirino;
613                                 nd = (struct dirent64 *)((char *)nd + ndlen);
614
615                                 /*
616                                  * free up space allocated for symlink
617                                  */
618                                 if (hd.sym_link != (char *)NULL) {
619                                         kmem_free(hd.sym_link,
620                                             (size_t)(hd.ext_size+1));
621                                         hd.sym_link = (char *)NULL;
622                                 }
623                         }
624                         offset += hdlen;
625                 }
626                 fbrelse(fbp, S_READ);
627         }
628
629         /*
630          * Got here for one of the following reasons:
631          *      1) outbuf is full (error == 0)
632          *      2) end of directory reached (error == 0)
633          *      3) error reading directory sector (error != 0)
634          *      4) directory entry crosses sector boundary (error == 0)
635          *
636          * If any directory entries have been copied, don't report
637          * case 4.  Instead, return the valid directory entries.
638          *
639          * If no entries have been copied, report the error.
640          * If case 4, this will be indistiguishable from EOF.
641          */
642 done:
643         ndlen = ((char *)nd - outbuf);
644         if (ndlen != 0) {
645                 error = uiomove(outbuf, (size_t)ndlen, UIO_READ, uiop);
646                 uiop->uio_loffset = offset;
647         }
648         kmem_free(dname, dname_size);
649         kmem_free(outbuf, bufsize);
650         if (eofp && error == 0)
651                 *eofp = (uiop->uio_loffset >= dirsiz);
652         return (error);
653 }
654
655 /*ARGSUSED2*/
```

```
656 static int
657 hsfs_fid(struct vnode *vp, struct fid *fidp, caller_context_t *ct)
658 {
659         struct hsnode *hp;
660         struct hsfid *fid;
661 
662         if (fidp->fid_len < (sizeof (*fid) - sizeof (fid->hf_len))) {
663                 fidp->fid_len = sizeof (*fid) - sizeof (fid->hf_len);
664                 return (ENOSPC);
665         }
666 
667         fid = (struct hsfid *)fidp;
668         fid->hf_len = sizeof (*fid) - sizeof (fid->hf_len);
669         hp = VTOH(vp);
670         mutex_enter(&hp->hs_contents_lock);
671         fid->hf_dir_lbn = hp->hs_dir_lbn;
672         fid->hf_dir_off = (ushort_t)hp->hs_dir_off;
673         fid->hf_ino = hp->hs_nodeid;
674         mutex_exit(&hp->hs_contents_lock);
675         return (0);
676 }
677 
678 /*ARGSUSED*/
679 static int
680 hsfs_open(struct vnode **vpp,
681         int flag,
682         struct cred *cred,
683         caller_context_t *ct)
684 {
685         return (0);
686 }
687 
688 /*ARGSUSED*/
689 static int
690 hsfs_close(
691         struct vnode *vp,
692         int flag,
693         int count,
694         offset_t offset,
695         struct cred *cred,
696         caller_context_t *ct)
697 {
698         (void) cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
699         cleanshares(vp, ttoproc(curthread)->p_pid);
700         return (0);
701 }
702 
703 /*ARGSUSED2*/
704 static int
705 hsfs_access(struct vnode *vp,
706         int mode,
707         int flags,
708         cred_t *cred,
709         caller_context_t *ct)
710 {
711         return (hs_access(vp, (mode_t)mode, cred));
712 }
713 
714 /*
715  * the seek time of a CD-ROM is very slow, and data transfer
716  * rate is even worse (max. 150K per sec).  The design
717  * decision is to reduce access to cd-rom as much as possible,
718  * and to transfer a sizable block (read-ahead) of data at a time.
719  * UFS style of read ahead one block at a time is not appropriate,
720  * and is not supported
721  */
```



```
722 
723 /*
724  * KLUSTSIZE should be a multiple of PAGESIZE and <= MAXPHYS.
725  */
726 #define KLUSTSIZE       (56 * 1024)
727 /* we don't support read ahead */
728 int hsfs_lostpage;      /* no. of times we lost original page */
729 
730 /*
731  * Used to prevent biodone() from releasing buf resources that
732  * we didn't allocate in quite the usual way.
733  */
734 /*ARGSUSED*/
735 int
736 hsfs_iodone(struct buf *bp)
737 {
738         sema_v(&bp->b_io);
739         return (0);
740 }
741 
742 /*
743  * The taskq thread that invokes the scheduling function to ensure
744  * that all readaheads are complete and cleans up the associated
745  * memory and releases the page lock.
746  */
747 void
748 hsfs_ra_task(void *arg)
749 {
750         struct hio_info *info = arg;
751         uint_t count;
752         struct buf *wbuf;
753 
754         ASSERT(info->pp != NULL);
755 
756         for (count = 0; count < info->bufsused; count++) {
757                 wbuf = &(info->bufs[count]);
758 
759                 DTRACE_PROBE1(hsfs_io_wait_ra, struct buf *, wbuf);
760                 while (sema_tryp(&(info->sema[count])) == 0) {
761                         if (hsched_invoke_strategy(info->fsp)) {
762                                 sema_p(&(info->sema[count]));
763                                 break;
764                         }
765                 }
766                 sema_destroy(&(info->sema[count]));
767                 DTRACE_PROBE1(hsfs_io_done_ra, struct buf *, wbuf);
768                 biofini(&(info->bufs[count]));
769         }
770         for (count = 0; count < info->bufsused; count++) {
771                 if (info->vas[count] != NULL) {
772                         ppmapout(info->vas[count]);
773                 }
774         }
775         kmem_free(info->vas, info->bufcnt * sizeof (caddr_t));
776         kmem_free(info->bufs, info->bufcnt * sizeof (struct buf));
777         kmem_free(info->sema, info->bufcnt * sizeof (ksema_t));
778 
779         pvn_read_done(info->pp, 0);
780         kmem_cache_free(hio_info_cache, info);
781 }
782 
783 /*
784  * Submit asynchronous readahead requests to the I/O scheduler
785  * depending on the number of pages to read ahead. These requests
786  * are asynchronous to the calling thread but I/O requests issued
787  * subsequently by other threads with higher LBNs must wait for
```

```
 * these readaheads to complete since we have a single ordered
 * I/O pipeline. Thus these readaheads are semi-asynchronous.
 * A TaskQ handles waiting for the readaheads to complete.
 *
 * This function is mostly a copy of hsfs_getapage but somewhat
 * simpler. A readahead request is aborted if page allocation
 * fails.
 */
/*ARGSUSED*/
static int
hsfs_getpage_ra(
        struct vnode *vp,
        u_offset_t off,
        struct seg *seg,
        caddr_t addr,
        struct hsnode *hp,
        struct hsfs *fsp,
        int     xarsiz,
        offset_t        bof,
        int     chunk_lbn_count,
        int     chunk_data_bytes)
{
        struct buf *bufs;
        caddr_t *vas;
        caddr_t va;
        struct page *pp, *searchp, *lastp;
        struct vnode *devvp;
        ulong_t byte_offset;
        size_t  io_len_tmp;
        uint_t  io_off, io_len;
        uint_t  xlen;
        uint_t  filsiz;
        uint_t  secsize;
        uint_t  bufcnt;
        uint_t  bufsused;
        uint_t  count;
        uint_t  io_end;
        uint_t  which_chunk_lbn;
        uint_t  offset_lbn;
        uint_t  offset_extra;
        offset_t        offset_bytes;
        uint_t  remaining_bytes;
        uint_t  extension;
        int     remainder;      /* must be signed */
        diskaddr_t driver_block;
        u_offset_t io_off_tmp;
        ksema_t *fio_done;
        struct hio_info *info;
        size_t len;

        ASSERT(fsp->hqueue != NULL);

        if (addr >= seg->s_base + seg->s_size) {
                return (-1);
        }

        devvp = fsp->hsfs_devvp;
        secsize = fsp->hsfs_vol.lbn_size;  /* bytes per logical block */

        /* file data size */
        filsiz = hp->hs_dirent.ext_size;

        if (off >= filsiz)
                return (0);

        extension = 0;
```



```
        pp = NULL;

        extension += hp->hs_ra_bytes;

        /*
         * Some CD writers (e.g. Kodak Photo CD writers)
         * create CDs in TAO mode and reserve tracks that
         * are not completely written. Some sectors remain
         * unreadable for this reason and give I/O errors.
         * Also, there's no point in reading sectors
         * we'll never look at.  So, if we're asked to go
         * beyond the end of a file, truncate to the length
         * of that file.
         *
         * Additionally, this behaviour is required by section
         * 6.4.5 of ISO 9660:1988(E).
         */
        len = MIN(extension ? extension : PAGESIZE, filsiz - off);

        /* A little paranoia */
        if (len <= 0)
                return (-1);

        /*
         * After all that, make sure we're asking for things in units
         * that bdev_strategy() will understand (see bug 4202551).
         */
        len = roundup(len, DEV_BSIZE);

        pp = pvn_read_kluster(vp, off, seg, addr, &io_off_tmp,
            &io_len_tmp, off, len, 1);

        if (pp == NULL) {
                hp->hs_num_contig = 0;
                hp->hs_ra_bytes = 0;
                hp->hs_prev_offset = 0;
                return (-1);
        }

        io_off = (uint_t)io_off_tmp;
        io_len = (uint_t)io_len_tmp;

        /* check for truncation */
        /*
         * xxx Clean up and return EIO instead?
         * xxx Ought to go to u_offset_t for everything, but we
         * xxx call lots of things that want uint_t arguments.
         */
        ASSERT(io_off == io_off_tmp);

        /*
         * get enough buffers for worst-case scenario
         * (i.e., no coalescing possible).
         */
        bufcnt = (len + secsize - 1) / secsize;
        bufs = kmem_alloc(bufcnt * sizeof (struct buf), KM_SLEEP);
        vas = kmem_alloc(bufcnt * sizeof (caddr_t), KM_SLEEP);

        /*
         * Allocate a array of semaphores since we are doing I/O
         * scheduling.
         */
        fio_done = kmem_alloc(bufcnt * sizeof (ksema_t), KM_SLEEP);

        /*
         * If our filesize is not an integer multiple of PAGESIZE,
```

```
920          * we zero that part of the last page that's between EOF and
921          * the PAGESIZE boundary.
922          */
923         xlen = io_len & PAGEOFFSET;
924         if (xlen != 0)
925                 pagezero(pp->p_prev, xlen, PAGESIZE - xlen);

927         DTRACE_PROBE2(hsfs_readahead, struct vnode *, vp, uint_t, io_len);

929         va = NULL;
930         lastp = NULL;
931         searchp = pp;
932         io_end = io_off + io_len;
933         for (count = 0, byte_offset = io_off;
934             byte_offset < io_end;
935             count++) {
936                 ASSERT(count < bufcnt);

938                 bioinit(&bufs[count]);
939                 bufs[count].b_edev = devvp->v_rdev;
940                 bufs[count].b_dev = cmpdev(devvp->v_rdev);
941                 bufs[count].b_flags = B_NOCACHE|B_BUSY|B_READ;
942                 bufs[count].b_iodone = hsfs_iodone;
943                 bufs[count].b_vp = vp;
944                 bufs[count].b_file = vp;

946                 /* Compute disk address for interleaving. */

948                 /* considered without skips */
949                 which_chunk_lbn = byte_offset / chunk_data_bytes;

951                 /* factor in skips */
952                 offset_lbn = which_chunk_lbn * chunk_lbn_count;

954                 /* convert to physical byte offset for lbn */
955                 offset_bytes = LBN_TO_BYTE(offset_lbn, vp->v_vfsp);

957                 /* don't forget offset into lbn */
958                 offset_extra = byte_offset % chunk_data_bytes;

960                 /* get virtual block number for driver */
961                 driver_block = lbtodb(bof + xarsiz
962                     + offset_bytes + offset_extra);

964                 if (lastp != searchp) {
965                         /* this branch taken first time through loop */
966                         va = vas[count] = ppmapin(searchp, PROT_WRITE,
967                             (caddr_t)-1);
968                         /* ppmapin() guarantees not to return NULL */
969                 } else {
970                         vas[count] = NULL;
971                 }

973                 bufs[count].b_un.b_addr = va + byte_offset % PAGESIZE;
974                 bufs[count].b_offset =
975                     (offset_t)(byte_offset - io_off + off);

977                 /*
978                  * We specifically use the b_lblkno member here
979                  * as even in the 32 bit world driver_block can
980                  * get very large in line with the ISO9660 spec.
981                  */

983                 bufs[count].b_lblkno = driver_block;

985                 remaining_bytes = ((which_chunk_lbn + 1) * chunk_data_bytes)
```



```
986                     - byte_offset;

988                 /*
989                  * remaining_bytes can't be zero, as we derived
990                  * which_chunk_lbn directly from byte_offset.
991                  */
992                 if ((remaining_bytes + byte_offset) < (off + len)) {
993                         /* coalesce-read the rest of the chunk */
994                         bufs[count].b_bcount = remaining_bytes;
995                 } else {
996                         /* get the final bits */
997                         bufs[count].b_bcount = off + len - byte_offset;
998                 }

1000                remainder = PAGESIZE - (byte_offset % PAGESIZE);
1001                if (bufs[count].b_bcount > remainder) {
1002                        bufs[count].b_bcount = remainder;
1003                }

1005                bufs[count].b_bufsize = bufs[count].b_bcount;
1006                if (((offset_t)byte_offset + bufs[count].b_bcount) >
1007                    HS_MAXFILEOFF) {
1008                        break;
1009                }
1010                byte_offset += bufs[count].b_bcount;

1012                /*
1013                 * We are scheduling I/O so we need to enqueue
1014                 * requests rather than calling bdev_strategy
1015                 * here. A later invocation of the scheduling
1016                 * function will take care of doing the actual
1017                 * I/O as it selects requests from the queue as
1018                 * per the scheduling logic.
1019                 */
1020                struct hio *hsio = kmem_cache_alloc(hio_cache,
1021                    KM_SLEEP);

1023                sema_init(&fio_done[count], 0, NULL,
1024                    SEMA_DEFAULT, NULL);
1025                hsio->bp = &bufs[count];
1026                hsio->sema = &fio_done[count];
1027                hsio->io_lblkno = bufs[count].b_lblkno;
1028                hsio->nblocks = howmany(hsio->bp->b_bcount,
1029                    DEV_BSIZE);

1031                /* used for deadline */
1032                hsio->io_timestamp = drv_hztousec(ddi_get_lbolt());

1034                /* for I/O coalescing */
1035                hsio->contig_chain = NULL;
1036                hsched_enqueue_io(fsp, hsio, 1);

1038                lwp_stat_update(LWP_STAT_INBLK, 1);
1039                lastp = searchp;
1040                if ((remainder - bufs[count].b_bcount) < 1) {
1041                        searchp = searchp->p_next;
1042                }
1043        }

1045        bufsused = count;
1046        info = kmem_cache_alloc(hio_info_cache, KM_SLEEP);
1047        info->bufs = bufs;
1048        info->vas = vas;
1049        info->sema = fio_done;
1050        info->bufsused = bufsused;
1051        info->bufcnt = bufcnt;
```

```
1052         info->fsp = fsp;
1053         info->pp = pp;

1055         (void) taskq_dispatch(fsp->hqueue->ra_task,
1056             hsfs_ra_task, info, KM_SLEEP);
1057         /*
1058          * The I/O locked pages are unlocked in our taskq thread.
1059          */
1060         return (0);
1061 }

1063 /*
1064  * Each file may have a different interleaving on disk.  This makes
1065  * things somewhat interesting.  The gist is that there are some
1066  * number of contiguous data sectors, followed by some other number
1067  * of contiguous skip sectors.  The sum of those two sets of sectors
1068  * defines the interleave size.  Unfortunately, it means that we generally
1069  * can't simply read N sectors starting at a given offset to satisfy
1070  * any given request.
1071  *
1072  * What we do is get the relevant memory pages via pvn_read_kluster(),
1073  * then stride through the interleaves, setting up a buf for each
1074  * sector that needs to be brought in.  Instead of kmem_alloc'ing
1075  * space for the sectors, though, we just point at the appropriate
1076  * spot in the relevant page for each of them.  This saves us a bunch
1077  * of copying.
1078  *
1079  * NOTICE: The code below in hsfs_getapage is mostly same as the code
1080  *         in hsfs_getpage_ra above (with some omissions). If you are
1081  *         making any change to this function, please also look at
1082  *         hsfs_getpage_ra.
1083  */
1084 /*ARGSUSED*/
1085 static int
1086 hsfs_getapage(
1087         struct vnode *vp,
1088         u_offset_t off,
1089         size_t len,
1090         uint_t *protp,
1091         struct page *pl[],
1092         size_t plsz,
1093         struct seg *seg,
1094         caddr_t addr,
1095         enum seg_rw rw,
1096         struct cred *cred)
1097 {
1098         struct hsnode *hp;
1099         struct hsfs *fsp;
1100         int     err;
1101         struct buf *bufs;
1102         caddr_t *vas;
1103         caddr_t va;
1104         struct page *pp, *searchp, *lastp;
1105         page_t  *pagefound;
1106         offset_t        bof;
1107         struct vnode *devvp;
1108         ulong_t byte_offset;
1109         size_t  io_len_tmp;
1110         uint_t  io_off, io_len;
1111         uint_t  xlen;
1112         uint_t  filsiz;
1113         uint_t  secsize;
1114         uint_t  bufcnt;
1115         uint_t  bufsused;
1116         uint_t  count;
1117         uint_t  io_end;
```



```
1118         uint_t  which_chunk_lbn;
1119         uint_t  offset_lbn;
1120         uint_t  offset_extra;
1121         offset_t        offset_bytes;
1122         uint_t  remaining_bytes;
1123         uint_t  extension;
1124         int     remainder;      /* must be signed */
1125         int     chunk_lbn_count;
1126         int     chunk_data_bytes;
1127         int     xarsiz;
1128         diskaddr_t driver_block;
1129         u_offset_t io_off_tmp;
1130         ksema_t *fio_done;
1131         int     calcdone;

1133         /*
1134          * We don't support asynchronous operation at the moment, so
1135          * just pretend we did it.  If the pages are ever actually
1136          * needed, they'll get brought in then.
1137          */
1138         if (pl == NULL)
1139                 return (0);

1141         hp = VTOH(vp);
1142         fsp = VFS_TO_HSFS(vp->v_vfsp);
1143         devvp = fsp->hsfs_devvp;
1144         secsize = fsp->hsfs_vol.lbn_size;  /* bytes per logical block */

1146         /* file data size */
1147         filsiz = hp->hs_dirent.ext_size;

1149         /* disk addr for start of file */
1150         bof = LBN_TO_BYTE((offset_t)hp->hs_dirent.ext_lbn, vp->v_vfsp);

1152         /* xarsiz byte must be skipped for data */
1153         xarsiz = hp->hs_dirent.xar_len << fsp->hsfs_vol.lbn_shift;

1155         /* how many logical blocks in an interleave (data+skip) */
1156         chunk_lbn_count = hp->hs_dirent.intlf_sz + hp->hs_dirent.intlf_sk;

1158         if (chunk_lbn_count == 0) {
1159                 chunk_lbn_count = 1;
1160         }

1162         /*
1163          * Convert interleaving size into bytes.  The zero case
1164          * (no interleaving) optimization is handled as a side-
1165          * effect of the read-ahead logic.
1166          */
1167         if (hp->hs_dirent.intlf_sz == 0) {
1168                 chunk_data_bytes = LBN_TO_BYTE(1, vp->v_vfsp);
1169                 /*
1170                  * Optimization: If our pagesize is a multiple of LBN
1171                  * bytes, we can avoid breaking up a page into individual
1172                  * lbn-sized requests.
1173                  */
1174                 if (PAGESIZE % chunk_data_bytes == 0) {
1175                         chunk_lbn_count = BYTE_TO_LBN(PAGESIZE, vp->v_vfsp);
1176                         chunk_data_bytes = PAGESIZE;
1177                 }
1178         } else {
1179                 chunk_data_bytes =
1180                     LBN_TO_BYTE(hp->hs_dirent.intlf_sz, vp->v_vfsp);
1181         }

1183 reread:
```

```
1184         err = 0;
1185         pagefound = 0;
1186         calcdone = 0;
1188         /*
1189          * Do some read-ahead.  This mostly saves us a bit of
1190          * system cpu time more than anything else when doing
1191          * sequential reads.  At some point, could do the
1192          * read-ahead asynchronously which might gain us something
1193          * on wall time, but it seems unlikely....
1194          *
1195          * We do the easy case here, which is to read through
1196          * the end of the chunk, minus whatever's at the end that
1197          * won't exactly fill a page.
1198          */
1199         if (hp->hs_ra_bytes > 0 && chunk_data_bytes != PAGESIZE) {
1200                 which_chunk_lbn = (off + len) / chunk_data_bytes;
1201                 extension = ((which_chunk_lbn + 1) * chunk_data_bytes) - off;
1202                 extension -= (extension % PAGESIZE);
1203         } else {
1204                 extension = roundup(len, PAGESIZE);
1205         }
1207         atomic_inc_64(&fsp->total_pages_requested);
1209         pp = NULL;
1210 again:
1211         /* search for page in buffer */
1212         if ((pagefound = page_exists(vp, off)) == 0) {
1213                 /*
1214                  * Need to really do disk IO to get the page.
1215                  */
1216                 if (!calcdone) {
1217                         extension += hp->hs_ra_bytes;
1219                         /*
1220                          * Some cd writers don't write sectors that aren't
1221                          * used. Also, there's no point in reading sectors
1222                          * we'll never look at.  So, if we're asked to go
1223                          * beyond the end of a file, truncate to the length
1224                          * of that file.
1225                          *
1226                          * Additionally, this behaviour is required by section
1227                          * 6.4.5 of ISO 9660:1988(E).
1228                          */
1229                         len = MIN(extension ? extension : PAGESIZE,
1230                             filsiz - off);
1232                         /* A little paranoia. */
1233                         ASSERT(len > 0);
1235                         /*
1236                          * After all that, make sure we're asking for things
1237                          * in units that bdev_strategy() will understand
1238                          * (see bug 4202551).
1239                          */
1240                         len = roundup(len, DEV_BSIZE);
1241                         calcdone = 1;
1242                 }
1244                 pp = pvn_read_kluster(vp, off, seg, addr, &io_off_tmp,
1245                     &io_len_tmp, off, len, 0);
1247                 if (pp == NULL) {
1248                         /*
1249                          * Pressure on memory, roll back readahead
```



```
1250                          */
1251                         hp->hs_num_contig = 0;
1252                         hp->hs_ra_bytes = 0;
1253                         hp->hs_prev_offset = 0;
1254                         goto again;
1255                 }
1257                 io_off = (uint_t)io_off_tmp;
1258                 io_len = (uint_t)io_len_tmp;
1260                 /* check for truncation */
1261                 /*
1262                  * xxx Clean up and return EIO instead?
1263                  * xxx Ought to go to u_offset_t for everything, but we
1264                  * xxx call lots of things that want uint_t arguments.
1265                  */
1266                 ASSERT(io_off == io_off_tmp);
1268                 /*
1269                  * get enough buffers for worst-case scenario
1270                  * (i.e., no coalescing possible).
1271                  */
1272                 bufcnt = (len + secsize - 1) / secsize;
1273                 bufs = kmem_zalloc(bufcnt * sizeof (struct buf), KM_SLEEP);
1274                 vas = kmem_alloc(bufcnt * sizeof (caddr_t), KM_SLEEP);
1276                 /*
1277                  * Allocate a array of semaphores if we are doing I/O
1278                  * scheduling.
1279                  */
1280                 if (fsp->hqueue != NULL)
1281                         fio_done = kmem_alloc(bufcnt * sizeof (ksema_t),
1282                             KM_SLEEP);
1283                 for (count = 0; count < bufcnt; count++) {
1284                         bioinit(&bufs[count]);
1285                         bufs[count].b_edev = devvp->v_rdev;
1286                         bufs[count].b_dev = cmpdev(devvp->v_rdev);
1287                         bufs[count].b_flags = B_NOCACHE|B_BUSY|B_READ;
1288                         bufs[count].b_iodone = hsfs_iodone;
1289                         bufs[count].b_vp = vp;
1290                         bufs[count].b_file = vp;
1291                 }
1293                 /*
1294                  * If our filesize is not an integer multiple of PAGESIZE,
1295                  * we zero that part of the last page that's between EOF and
1296                  * the PAGESIZE boundary.
1297                  */
1298                 xlen = io_len & PAGEOFFSET;
1299                 if (xlen != 0)
1300                         pagezero(pp->p_prev, xlen, PAGESIZE - xlen);
1302                 va = NULL;
1303                 lastp = NULL;
1304                 searchp = pp;
1305                 io_end = io_off + io_len;
1306                 for (count = 0, byte_offset = io_off;
1307                     byte_offset < io_end; count++) {
1308                         ASSERT(count < bufcnt);
1310                         /* Compute disk address for interleaving. */
1312                         /* considered without skips */
1313                         which_chunk_lbn = byte_offset / chunk_data_bytes;
1315                         /* factor in skips */
```

```
1316                        offset_lbn = which_chunk_lbn * chunk_lbn_count;

1318                        /* convert to physical byte offset for lbn */
1319                        offset_bytes = LBN_TO_BYTE(offset_lbn, vp->v_vfsp);

1321                        /* don't forget offset into lbn */
1322                        offset_extra = byte_offset % chunk_data_bytes;

1324                        /* get virtual block number for driver */
1325                        driver_block =
1326                            lbtodb(bof + xarsiz + offset_bytes + offset_extra);

1328                        if (lastp != searchp) {
1329                                /* this branch taken first time through loop */
1330                                va = vas[count] =
1331                                    ppmapin(searchp, PROT_WRITE, (caddr_t)-1);
1332                                /* ppmapin() guarantees not to return NULL */
1333                        } else {
1334                                vas[count] = NULL;
1335                        }

1337                        bufs[count].b_un.b_addr = va + byte_offset % PAGESIZE;
1338                        bufs[count].b_offset =
1339                            (offset_t)(byte_offset - io_off + off);

1341                        /*
1342                         * We specifically use the b_lblkno member here
1343                         * as even in the 32 bit world driver_block can
1344                         * get very large in line with the ISO9660 spec.
1345                         */

1347                        bufs[count].b_lblkno = driver_block;

1349                        remaining_bytes =
1350                            ((which_chunk_lbn + 1) * chunk_data_bytes)
1351                            - byte_offset;

1353                        /*
1354                         * remaining_bytes can't be zero, as we derived
1355                         * which_chunk_lbn directly from byte_offset.
1356                         */
1357                        if ((remaining_bytes + byte_offset) < (off + len)) {
1358                                /* coalesce-read the rest of the chunk */
1359                                bufs[count].b_bcount = remaining_bytes;
1360                        } else {
1361                                /* get the final bits */
1362                                bufs[count].b_bcount = off + len - byte_offset;
1363                        }

1365                        /*
1366                         * It would be nice to do multiple pages'
1367                         * worth at once here when the opportunity
1368                         * arises, as that has been shown to improve
1369                         * our wall time.  However, to do that
1370                         * requires that we use the pageio subsystem,
1371                         * which doesn't mix well with what we're
1372                         * already using here.  We can't use pageio
1373                         * all the time, because that subsystem
1374                         * assumes that a page is stored in N
1375                         * contiguous blocks on the device.
1376                         * Interleaving violates that assumption.
1377                         *
1378                         * Update: This is now not so big a problem
1379                         * because of the I/O scheduler sitting below
1380                         * that can re-order and coalesce I/O requests.
1381                         */
```



```
1383                        remainder = PAGESIZE - (byte_offset % PAGESIZE);
1384                        if (bufs[count].b_bcount > remainder) {
1385                                bufs[count].b_bcount = remainder;
1386                        }

1388                        bufs[count].b_bufsize = bufs[count].b_bcount;
1389                        if (((offset_t)byte_offset + bufs[count].b_bcount) >
1390                            HS_MAXFILEOFF) {
1391                                break;
1392                        }
1393                        byte_offset += bufs[count].b_bcount;

1395                        if (fsp->hqueue == NULL) {
1396                                (void) bdev_strategy(&bufs[count]);

1398                        } else {
1399                                /*
1400                                 * We are scheduling I/O so we need to enqueue
1401                                 * requests rather than calling bdev_strategy
1402                                 * here. A later invocation of the scheduling
1403                                 * function will take care of doing the actual
1404                                 * I/O as it selects requests from the queue as
1405                                 * per the scheduling logic.
1406                                 */
1407                                struct hio *hsio = kmem_cache_alloc(hio_cache,
1408                                    KM_SLEEP);

1410                                sema_init(&fio_done[count], 0, NULL,
1411                                    SEMA_DEFAULT, NULL);
1412                                hsio->bp = &bufs[count];
1413                                hsio->sema = &fio_done[count];
1414                                hsio->io_lblkno = bufs[count].b_lblkno;
1415                                hsio->nblocks = howmany(hsio->bp->b_bcount,
1416                                    DEV_BSIZE);

1418                                /* used for deadline */
1419                                hsio->io_timestamp =
1420                                    drv_hztousec(ddi_get_lbolt());

1422                                /* for I/O coalescing */
1423                                hsio->contig_chain = NULL;
1424                                hsched_enqueue_io(fsp, hsio, 0);
1425                        }

1427                        lwp_stat_update(LWP_STAT_INBLK, 1);
1428                        lastp = searchp;
1429                        if ((remainder - bufs[count].b_bcount) < 1) {
1430                                searchp = searchp->p_next;
1431                        }
1432                }

1434                bufsused = count;
1435                /* Now wait for everything to come in */
1436                if (fsp->hqueue == NULL) {
1437                        for (count = 0; count < bufsused; count++) {
1438                                if (err == 0) {
1439                                        err = biowait(&bufs[count]);
1440                                } else
1441                                        (void) biowait(&bufs[count]);
1442                        }
1443                } else {
1444                        for (count = 0; count < bufsused; count++) {
1445                                struct buf *wbuf;

1447                                /*
```

```
1448                                 * Invoke scheduling function till our buf
1449                                 * is processed. In doing this it might
1450                                 * process bufs enqueued by other threads
1451                                 * which is good.
1452                                 */
1453                                wbuf = &bufs[count];
1454                                DTRACE_PROBE1(hsfs_io_wait, struct buf *, wbuf);
1455                                while (sema_tryp(&fio_done[count]) == 0) {
1456                                        /*
1457                                         * hsched_invoke_strategy will return 1
1458                                         * if the I/O queue is empty. This means
1459                                         * that there is another thread who has
1460                                         * issued our buf and is waiting. So we
1461                                         * just block instead of spinning.
1462                                         */
1463                                        if (hsched_invoke_strategy(fsp)) {
1464                                                sema_p(&fio_done[count]);
1465                                                break;
1466                                        }
1467                                }
1468                                sema_destroy(&fio_done[count]);
1469                                DTRACE_PROBE1(hsfs_io_done, struct buf *, wbuf);

1471                                if (err == 0) {
1472                                        err = geterror(wbuf);
1473                                }
1474                        }
1475                        kmem_free(fio_done, bufcnt * sizeof (ksema_t));
1476                }

1478                /* Don't leak resources */
1479                for (count = 0; count < bufcnt; count++) {
1480                        biofini(&bufs[count]);
1481                        if (count < bufsused && vas[count] != NULL) {
1482                                ppmapout(vas[count]);
1483                        }
1484                }

1486                kmem_free(vas, bufcnt * sizeof (caddr_t));
1487                kmem_free(bufs, bufcnt * sizeof (struct buf));
1488        }

1490        if (err) {
1491                pvn_read_done(pp, B_ERROR);
1492                return (err);
1493        }

1495        /*
1496         * Lock the requested page, and the one after it if possible.
1497         * Don't bother if our caller hasn't given us a place to stash
1498         * the page pointers, since otherwise we'd lock pages that would
1499         * never get unlocked.
1500         */
1501        if (pagefound) {
1502                int index;
1503                ulong_t soff;

1505                /*
1506                 * Make sure it's in memory before we say it's here.
1507                 */
1508                if ((pp = page_lookup(vp, off, SE_SHARED)) == NULL) {
1509                        hsfs_lostpage++;
1510                        goto reread;
1511                }

1513                pl[0] = pp;
```



```
1514                index = 1;
1515                atomic_inc_64(&fsp->cache_read_pages);

1517                /*
1518                 * Try to lock the next page, if it exists, without
1519                 * blocking.
1520                 */
1521                plsz -= PAGESIZE;
1522                /* LINTED (plsz is unsigned) */
1523                for (soff = off + PAGESIZE; plsz > 0;
1524                    soff += PAGESIZE, plsz -= PAGESIZE) {
1525                        pp = page_lookup_nowait(vp, (u_offset_t)soff,
1526                            SE_SHARED);
1527                        if (pp == NULL)
1528                                break;
1529                        pl[index++] = pp;
1530                }
1531                pl[index] = NULL;

1533                /*
1534                 * Schedule a semi-asynchronous readahead if we are
1535                 * accessing the last cached page for the current
1536                 * file.
1537                 *
1538                 * Doing this here means that readaheads will be
1539                 * issued only if cache-hits occur. This is an advantage
1540                 * since cache-hits would mean that readahead is giving
1541                 * the desired benefit. If cache-hits do not occur there
1542                 * is no point in reading ahead of time - the system
1543                 * is loaded anyway.
1544                 */
1545                if (fsp->hqueue != NULL &&
1546                    hp->hs_prev_offset - off == PAGESIZE &&
1547                    hp->hs_prev_offset < filsiz &&
1548                    hp->hs_ra_bytes > 0 &&
1549                    !page_exists(vp, hp->hs_prev_offset)) {
1550                        (void) hsfs_getpage_ra(vp, hp->hs_prev_offset, seg,
1551                            addr + PAGESIZE, hp, fsp, xarsiz, bof,
1552                            chunk_lbn_count, chunk_data_bytes);
1553                }

1555                return (0);
1556        }

1558        if (pp != NULL) {
1559                pvn_plist_init(pp, pl, plsz, off, io_len, rw);
1560        }

1562        return (err);
1563 }

1565 /*ARGSUSED*/
1566 static int
1567 hsfs_getpage(
1568         struct vnode *vp,
1569         offset_t off,
1570         size_t len,
1571         uint_t *protp,
1572         struct page *pl[],
1573         size_t plsz,
1574         struct seg *seg,
1575         caddr_t addr,
1576         enum seg_rw rw,
1577         struct cred *cred,
1578         caller_context_t *ct)
1579 {
```

```
  25         int err;
1580         uint_t filsiz;
1581         struct hsfs *fsp;
1582         struct hsnode *hp;

1584         fsp = VFS_TO_HSFS(vp->v_vfsp);
1585         hp = VTOH(vp);

1587         /* does not support write */
1588         if (rw == S_WRITE) {
1589                 return (EROFS);
1590         }

1592         if (vp->v_flag & VNOMAP) {
1593                 return (ENOSYS);
1594         }

1596         ASSERT(off <= HS_MAXFILEOFF);

1598         /*
1599          * Determine file data size for EOF check.
1600          */
1601         filsiz = hp->hs_dirent.ext_size;
1602         if ((off + len) > (offset_t)(filsiz + PAGEOFFSET) && seg != segkmap)
1603                 return (EFAULT);        /* beyond EOF */

1605         /*
1606          * Async Read-ahead computation.
1607          * This attempts to detect sequential access pattern and
1608          * enables reading extra pages ahead of time.
1609          */
1610         if (fsp->hqueue != NULL) {
1611                 /*
1612                  * This check for sequential access also takes into
1613                  * account segmap weirdness when reading in chunks
1614                  * less than the segmap size of 8K.
1615                  */
1616                 if (hp->hs_prev_offset == off || (off <
1617                     hp->hs_prev_offset && off + MAX(len, PAGESIZE)
1618                     >= hp->hs_prev_offset)) {
1619                         if (hp->hs_num_contig <
1620                             (seq_contig_requests - 1)) {
1621                                 hp->hs_num_contig++;

1623                         } else {
1624                                 /*
1625                                  * We increase readahead quantum till
1626                                  * a predefined max. max_readahead_bytes
1627                                  * is a multiple of PAGESIZE.
1628                                  */
1629                                 if (hp->hs_ra_bytes <
1630                                     fsp->hqueue->max_ra_bytes) {
1631                                         hp->hs_ra_bytes += PAGESIZE;
1632                                 }
1633                         }
1634                 } else {
1635                         /*
1636                          * Not contiguous so reduce read ahead counters.
1637                          */
1638                         if (hp->hs_ra_bytes > 0)
1639                                 hp->hs_ra_bytes -= PAGESIZE;

1641                         if (hp->hs_ra_bytes <= 0) {
1642                                 hp->hs_ra_bytes = 0;
1643                                 if (hp->hs_num_contig > 0)
1644                                         hp->hs_num_contig--;
```



```
1645                         }
1646                 }
1647                 /*
1648                  * Length must be rounded up to page boundary.
1649                  * since we read in units of pages.
1650                  */
1651                 hp->hs_prev_offset = off + roundup(len, PAGESIZE);
1652                 DTRACE_PROBE1(hsfs_compute_ra, struct hsnode *, hp);
1653         }
1654         if (protp != NULL)
1655                 *protp = PROT_ALL;

1657         return (pvn_getpages(hsfs_getapage, vp, off, len, protp, pl, plsz,
1658             seg, addr, rw, cred));
 103         if (len <= PAGESIZE)
 104                 err = hsfs_getapage(vp, (u_offset_t)off, len, protp, pl, plsz,
 105                     seg, addr, rw, cred);
 106         else
 107                 err = pvn_getpages(hsfs_getapage, vp, off, len, protp,
 108                     pl, plsz, seg, addr, rw, cred);

 110         return (err);
1659 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  171811 Thu Jan  8 09:14:34 2015
new/usr/src/uts/common/fs/nfs/nfs3_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 *      Copyright (c) 1983,1984,1985,1986,1987,1988,1989 AT&T.
 *      All rights reserved.
 */

/*
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/systm.h>
#include <sys/cred.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/file.h>
#include <sys/filio.h>
#include <sys/uio.h>
#include <sys/buf.h>
#include <sys/mman.h>
#include <sys/pathname.h>
#include <sys/dirent.h>
#include <sys/debug.h>
#include <sys/vmsystm.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/swap.h>
#include <sys/errno.h>
#include <sys/strsubr.h>
#include <sys/sysmacros.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
```



```
#include <sys/pathconf.h>
#include <sys/utsname.h>
#include <sys/dnlc.h>
#include <sys/acl.h>
#include <sys/systeminfo.h>
#include <sys/atomic.h>
#include <sys/policy.h>
#include <sys/sdt.h>
#include <sys/zone.h>

#include <rpc/types.h>
#include <rpc/auth.h>
#include <rpc/clnt.h>
#include <rpc/rpc_rdma.h>

#include <nfs/nfs.h>
#include <nfs/nfs_clnt.h>
#include <nfs/rnode.h>
#include <nfs/nfs_acl.h>
#include <nfs/lm.h>

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

#include <fs/fs_subr.h>

#include <sys/ddi.h>

static int      nfs3_rdwrlbn(vnode_t *, page_t *, u_offset_t, size_t, int,
                        cred_t *);
static int      nfs3write(vnode_t *, caddr_t, u_offset_t, int, cred_t *,
                        stable_how *);
static int      nfs3read(vnode_t *, caddr_t, offset_t, int, size_t *, cred_t *);
static int      nfs3setattr(vnode_t *, struct vattr *, int, cred_t *);
static int      nfs3_accessx(void *, int, cred_t *);
static int      nfs3lookup_dnlc(vnode_t *, char *, vnode_t **, cred_t *);
static int      nfs3lookup_otw(vnode_t *, char *, vnode_t **, cred_t *, int);
static int      nfs3create(vnode_t *, char *, struct vattr *, enum vcexcl,
                        int, vnode_t **, cred_t *, int);
static int      nfs3excl_create_settimes(vnode_t *, struct vattr *, cred_t *);
static int      nfs3mknod(vnode_t *, char *, struct vattr *, enum vcexcl,
                        int, vnode_t **, cred_t *);
static int      nfs3rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
                        caller_context_t *);
static int      do_nfs3readdir(vnode_t *, rddir_cache *, cred_t *);
static void     nfs3readdir(vnode_t *, rddir_cache *, cred_t *);
static void     nfs3readdirplus(vnode_t *, rddir_cache *, cred_t *);
static int      nfs3_bio(struct buf *, stable_how *, cred_t *);
static int      nfs3_getapage(vnode_t *, u_offset_t, size_t, uint_t *,
                        page_t *[], size_t, struct seg *, caddr_t,
                        enum seg_rw, cred_t *);
static void     nfs3_readahead(vnode_t *, u_offset_t, caddr_t, struct seg *,
                        cred_t *);
static int      nfs3_sync_putapage(vnode_t *, page_t *, u_offset_t, size_t,
                        int, cred_t *);
static int      nfs3_sync_pageio(vnode_t *, page_t *, u_offset_t, size_t,
                        int, cred_t *);
static int      nfs3_commit(vnode_t *, offset3, count3, cred_t *);
static void     nfs3_set_mod(vnode_t *);
static void     nfs3_get_commit(vnode_t *);
```

```
128 static void     nfs3_get_commit_range(vnode_t *, u_offset_t, size_t);
129 static int      nfs3_putpage_commit(vnode_t *, offset_t, size_t, cred_t *);
130 static int      nfs3_commit_vp(vnode_t *, u_offset_t, size_t,  cred_t *);
131 static int      nfs3_sync_commit(vnode_t *, page_t *, offset3, count3,
132                         cred_t *);
133 static void     nfs3_async_commit(vnode_t *, page_t *, offset3, count3,
134                         cred_t *);
135 static void     nfs3_delmap_callback(struct as *, void *, uint_t);

137 /*
138  * Error flags used to pass information about certain special errors
139  * which need to be handled specially.
140  */
141 #define NFS_EOF                 -98
142 #define NFS_VERF_MISMATCH       -97

144 /* ALIGN64 aligns the given buffer and adjust buffer size to 64 bit */
145 #define ALIGN64(x, ptr, sz)                                             \
146         x = ((uintptr_t)(ptr)) & (sizeof (uint64_t) - 1);               \
147         if (x) {                                                        \
148                 x = sizeof (uint64_t) - (x);                            \
149                 sz -= (x);                                              \
150                 ptr += (x);                                             \
151         }

153 /*
154  * These are the vnode ops routines which implement the vnode interface to
155  * the networked file system.  These routines just take their parameters,
156  * make them look networkish by putting the right info into interface structs,
157  * and then calling the appropriate remote routine(s) to do the work.
158  *
159  * Note on directory name lookup cacheing:  If we detect a stale fhandle,
160  * we purge the directory cache relative to that vnode.  This way, the
161  * user won't get burned by the cache repeatedly.  See <nfs/rnode.h> for
162  * more details on rnode locking.
163  */

165 static int      nfs3_open(vnode_t **, int, cred_t *, caller_context_t *);
166 static int      nfs3_close(vnode_t *, int, int, offset_t, cred_t *,
167                         caller_context_t *);
168 static int      nfs3_read(vnode_t *, struct uio *, int, cred_t *,
169                         caller_context_t *);
170 static int      nfs3_write(vnode_t *, struct uio *, int, cred_t *,
171                         caller_context_t *);
172 static int      nfs3_ioctl(vnode_t *, int, intptr_t, int, cred_t *, int *,
173                         caller_context_t *);
174 static int      nfs3_getattr(vnode_t *, struct vattr *, int, cred_t *,
175                         caller_context_t *);
176 static int      nfs3_setattr(vnode_t *, struct vattr *, int, cred_t *,
177                         caller_context_t *);
178 static int      nfs3_access(vnode_t *, int, int, cred_t *, caller_context_t *);
179 static int      nfs3_readlink(vnode_t *, struct uio *, cred_t *,
180                         caller_context_t *);
181 static int      nfs3_fsync(vnode_t *, int, cred_t *, caller_context_t *);
182 static void     nfs3_inactive(vnode_t *, cred_t *, caller_context_t *);
183 static int      nfs3_lookup(vnode_t *, char *, vnode_t **,
184                         struct pathname *, int, vnode_t *, cred_t *,
185                         caller_context_t *, int *, pathname_t *);
186 static int      nfs3_create(vnode_t *, char *, struct vattr *, enum vcexcl,
187                         int, vnode_t **, cred_t *, int, caller_context_t *,
188                         vsecattr_t *);
189 static int      nfs3_remove(vnode_t *, char *, cred_t *, caller_context_t *,
190                         int);
191 static int      nfs3_link(vnode_t *, vnode_t *, char *, cred_t *,
192                         caller_context_t *, int);
193 static int      nfs3_rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
```



```
194                         caller_context_t *, int);
195 static int      nfs3_mkdir(vnode_t *, char *, struct vattr *, vnode_t **,
196                         cred_t *, caller_context_t *, int, vsecattr_t *);
197 static int      nfs3_rmdir(vnode_t *, char *, vnode_t *, cred_t *,
198                         caller_context_t *, int);
199 static int      nfs3_symlink(vnode_t *, char *, struct vattr *, char *,
200                         cred_t *, caller_context_t *, int);
201 static int      nfs3_readdir(vnode_t *, struct uio *, cred_t *, int *,
202                         caller_context_t *, int);
203 static int      nfs3_fid(vnode_t *, fid_t *, caller_context_t *);
204 static int      nfs3_rwlock(vnode_t *, int, caller_context_t *);
205 static void     nfs3_rwunlock(vnode_t *, int, caller_context_t *);
206 static int      nfs3_seek(vnode_t *, offset_t, offset_t *, caller_context_t *);
207 static int      nfs3_getpage(vnode_t *, offset_t, size_t, uint_t *,
208                         page_t *[], size_t, struct seg *, caddr_t,
209                         enum seg_rw, cred_t *, caller_context_t *);
210 static int      nfs3_putpage(vnode_t *, offset_t, size_t, int, cred_t *,
211                         caller_context_t *);
212 static int      nfs3_map(vnode_t *, offset_t, struct as *, caddr_t *, size_t,
213                         uchar_t, uchar_t, uint_t, cred_t *, caller_context_t *);
214 static int      nfs3_addmap(vnode_t *, offset_t, struct as *, caddr_t, size_t,
215                         uchar_t, uchar_t, uint_t, cred_t *, caller_context_t *);
216 static int      nfs3_frlock(vnode_t *, int, struct flock64 *, int, offset_t,
217                         struct flk_callback *, cred_t *, caller_context_t *);
218 static int      nfs3_space(vnode_t *, int, struct flock64 *, int, offset_t,
219                         cred_t *, caller_context_t *);
220 static int      nfs3_realvp(vnode_t *, vnode_t **, caller_context_t *);
221 static int      nfs3_delmap(vnode_t *, offset_t, struct as *, caddr_t, size_t,
222                         uint_t, uint_t, uint_t, cred_t *, caller_context_t *);
223 static int      nfs3_pathconf(vnode_t *, int, ulong_t *, cred_t *,
224                         caller_context_t *);
225 static int      nfs3_pageio(vnode_t *, page_t *, u_offset_t, size_t, int,
226                         cred_t *, caller_context_t *);
227 static void     nfs3_dispose(vnode_t *, page_t *, int, int, cred_t *,
228                         caller_context_t *);
229 static int      nfs3_setsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
230                         caller_context_t *);
231 static int      nfs3_getsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
232                         caller_context_t *);
233 static int      nfs3_shrlock(vnode_t *, int, struct shrlock *, int, cred_t *,
234                         caller_context_t *);

236 struct vnodeops *nfs3_vnodeops;

238 const fs_operation_def_t nfs3_vnodeops_template[] = {
239         VOPNAME_OPEN,           { .vop_open = nfs3_open },
240         VOPNAME_CLOSE,          { .vop_close = nfs3_close },
241         VOPNAME_READ,           { .vop_read = nfs3_read },
242         VOPNAME_WRITE,          { .vop_write = nfs3_write },
243         VOPNAME_IOCTL,          { .vop_ioctl = nfs3_ioctl },
244         VOPNAME_GETATTR,        { .vop_getattr = nfs3_getattr },
245         VOPNAME_SETATTR,        { .vop_setattr = nfs3_setattr },
246         VOPNAME_ACCESS,         { .vop_access = nfs3_access },
247         VOPNAME_LOOKUP,         { .vop_lookup = nfs3_lookup },
248         VOPNAME_CREATE,         { .vop_create = nfs3_create },
249         VOPNAME_REMOVE,         { .vop_remove = nfs3_remove },
250         VOPNAME_LINK,           { .vop_link = nfs3_link },
251         VOPNAME_RENAME,         { .vop_rename = nfs3_rename },
252         VOPNAME_MKDIR,          { .vop_mkdir = nfs3_mkdir },
253         VOPNAME_RMDIR,          { .vop_rmdir = nfs3_rmdir },
254         VOPNAME_READDIR,        { .vop_readdir = nfs3_readdir },
255         VOPNAME_SYMLINK,        { .vop_symlink = nfs3_symlink },
256         VOPNAME_READLINK,       { .vop_readlink = nfs3_readlink },
257         VOPNAME_FSYNC,          { .vop_fsync = nfs3_fsync },
258         VOPNAME_INACTIVE,       { .vop_inactive = nfs3_inactive },
259         VOPNAME_FID,            { .vop_fid = nfs3_fid },
```

```
260         VOPNAME_RWLOCK,         { .vop_rwlock = nfs3_rwlock },
261         VOPNAME_RWUNLOCK,       { .vop_rwunlock = nfs3_rwunlock },
262         VOPNAME_SEEK,           { .vop_seek = nfs3_seek },
263         VOPNAME_FRLOCK,         { .vop_frlock = nfs3_frlock },
264         VOPNAME_SPACE,          { .vop_space = nfs3_space },
265         VOPNAME_REALVP,         { .vop_realvp = nfs3_realvp },
266         VOPNAME_GETPAGE,        { .vop_getpage = nfs3_getpage },
267         VOPNAME_PUTPAGE,        { .vop_putpage = nfs3_putpage },
268         VOPNAME_MAP,            { .vop_map = nfs3_map },
269         VOPNAME_ADDMAP,         { .vop_addmap = nfs3_addmap },
270         VOPNAME_DELMAP,         { .vop_delmap = nfs3_delmap },
271         /* no separate nfs3_dump */
272         VOPNAME_DUMP,           { .vop_dump = nfs_dump },
273         VOPNAME_PATHCONF,       { .vop_pathconf = nfs3_pathconf },
274         VOPNAME_PAGEIO,         { .vop_pageio = nfs3_pageio },
275         VOPNAME_DISPOSE,        { .vop_dispose = nfs3_dispose },
276         VOPNAME_SETSECATTR,     { .vop_setsecattr = nfs3_setsecattr },
277         VOPNAME_GETSECATTR,     { .vop_getsecattr = nfs3_getsecattr },
278         VOPNAME_SHRLOCK,        { .vop_shrlock = nfs3_shrlock },
279         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
280         NULL,                   NULL
281 };

283 /*
284  * XXX:  This is referenced in modstubs.s
285  */
286 struct vnodeops *
287 nfs3_getvnodeops(void)
288 {
289         return (nfs3_vnodeops);
290 }

292 /* ARGSUSED */
293 static int
294 nfs3_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
295 {
296         int error;
297         struct vattr va;
298         rnode_t *rp;
299         vnode_t *vp;

301         vp = *vpp;
302         if (nfs_zone() != VTOMI(vp)->mi_zone)
303                 return (EIO);
304         rp = VTOR(vp);
305         mutex_enter(&rp->r_statelock);
306         if (rp->r_cred == NULL) {
307                 crhold(cr);
308                 rp->r_cred = cr;
309         }
310         mutex_exit(&rp->r_statelock);

312         /*
313          * If there is no cached data or if close-to-open
314          * consistency checking is turned off, we can avoid
315          * the over the wire getattr.  Otherwise, if the
316          * file system is mounted readonly, then just verify
317          * the caches are up to date using the normal mechanism.
318          * Else, if the file is not mmap'd, then just mark
319          * the attributes as timed out.  They will be refreshed
320          * and the caches validated prior to being used.
321          * Else, the file system is mounted writeable so
322          * force an over the wire GETATTR in order to ensure
323          * that all cached data is valid.
324          */
325         if (vp->v_count > 1 ||
```



```
326             ((vn_has_cached_data(vp) || HAVE_RDDIR_CACHE(rp)) &&
327             !(VTOMI(vp)->mi_flags & MI_NOCTO))) {
328                 if (vn_is_readonly(vp))
329                         error = nfs3_validate_caches(vp, cr);
330                 else if (rp->r_mapcnt == 0 && vp->v_count == 1) {
331                         PURGE_ATTRCACHE(vp);
332                         error = 0;
333                 } else {
334                         va.va_mask = AT_ALL;
335                         error = nfs3_getattr_otw(vp, &va, cr);
336                 }
337         } else
338                 error = 0;

340         return (error);
341 }

343 /* ARGSUSED */
344 static int
345 nfs3_close(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
346                 caller_context_t *ct)
347 {
348         rnode_t *rp;
349         int error;
350         struct vattr va;

352         /*
353          * zone_enter(2) prevents processes from changing zones with NFS files
354          * open; if we happen to get here from the wrong zone we can't do
355          * anything over the wire.
356          */
357         if (VTOMI(vp)->mi_zone != nfs_zone()) {
358                 /*
359                  * We could attempt to clean up locks, except we're sure
360                  * that the current process didn't acquire any locks on
361                  * the file: any attempt to lock a file belong to another zone
362                  * will fail, and one can't lock an NFS file and then change
363                  * zones, as that fails too.
364                  *
365                  * Returning an error here is the sane thing to do.  A
366                  * subsequent call to VN_RELE() which translates to a
367                  * nfs3_inactive() will clean up state: if the zone of the
368                  * vnode's origin is still alive and kicking, an async worker
369                  * thread will handle the request (from the correct zone), and
370                  * everything (minus the commit and final nfs3_getattr_otw()
371                  * call) should be OK. If the zone is going away
372                  * nfs_async_inactive() will throw away cached pages inline.
373                  */
374                 return (EIO);
375         }

377         /*
378          * If we are using local locking for this filesystem, then
379          * release all of the SYSV style record locks.  Otherwise,
380          * we are doing network locking and we need to release all
381          * of the network locks.  All of the locks held by this
382          * process on this file are released no matter what the
383          * incoming reference count is.
384          */
385         if (VTOMI(vp)->mi_flags & MI_LLOCK) {
386                 cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
387                 cleanshares(vp, ttoproc(curthread)->p_pid);
388         } else
389                 nfs_lockrelease(vp, flag, offset, cr);

391         if (count > 1)
```

```
392                 return (0);

394         /*
395          * If the file has been 'unlinked', then purge the
396          * DNLC so that this vnode will get reycled quicker
397          * and the .nfs* file on the server will get removed.
398          */
399         rp = VTOR(vp);
400         if (rp->r_unldvp != NULL)
401                 dnlc_purge_vp(vp);

403         /*
404          * If the file was open for write and there are pages,
405          * then if the file system was mounted using the "no-close-
406          *      to-open" semantics, then start an asynchronous flush
407          *      of the all of the pages in the file.
408          * else the file system was not mounted using the "no-close-
409          *      to-open" semantics, then do a synchronous flush and
410          *      commit of all of the dirty and uncommitted pages.
411          *
412          * The asynchronous flush of the pages in the "nocto" path
413          * mostly just associates a cred pointer with the rnode so
414          * writes which happen later will have a better chance of
415          * working.  It also starts the data being written to the
416          * server, but without unnecessarily delaying the application.
417          */
418         if ((flag & FWRITE) && vn_has_cached_data(vp)) {
419                 if (VTOMI(vp)->mi_flags & MI_NOCTO) {
420                         error = nfs3_putpage(vp, (offset_t)0, 0, B_ASYNC,
421                             cr, ct);
422                         if (error == EAGAIN)
423                                 error = 0;
424                 } else
425                         error = nfs3_putpage_commit(vp, (offset_t)0, 0, cr);
426                 if (!error) {
427                         mutex_enter(&rp->r_statelock);
428                         error = rp->r_error;
429                         rp->r_error = 0;
430                         mutex_exit(&rp->r_statelock);
431                 }
432         } else {
433                 mutex_enter(&rp->r_statelock);
434                 error = rp->r_error;
435                 rp->r_error = 0;
436                 mutex_exit(&rp->r_statelock);
437         }

439         /*
440          * If RWRITEATTR is set, then issue an over the wire GETATTR to
441          * refresh the attribute cache with a set of attributes which
442          * weren't returned from a WRITE.  This will enable the close-
443          * to-open processing to work.
444          */
445         if (rp->r_flags & RWRITEATTR)
446                 (void) nfs3_getattr_otw(vp, &va, cr);

448         return (error);
449 }

451 /* ARGSUSED */
452 static int
453 nfs3_directio_read(vnode_t *vp, struct uio *uiop, cred_t *cr)
454 {
455         mntinfo_t *mi;
456         READ3args args;
457         READ3uiores res;
```



```
458         int tsize;
459         offset_t offset;
460         ssize_t count;
461         int error;
462         int douprintf;
463         failinfo_t fi;
464         char *sv_hostname;

466         mi = VTOMI(vp);
467         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);
468         sv_hostname = VTOR(vp)->r_server->sv_hostname;

470         douprintf = 1;
471         args.file = *VTOFH3(vp);
472         fi.vp = vp;
473         fi.fhp = (caddr_t)&args.file;
474         fi.copyproc = nfs3copyfh;
475         fi.lookupproc = nfs3lookup;
476         fi.xattrdirproc = acl_getxattrdir3;

478         res.uiop = uiop;

480         res.wlist = NULL;

482         offset = uiop->uio_loffset;
483         count = uiop->uio_resid;

485         do {
486                 if (mi->mi_io_kstats) {
487                         mutex_enter(&mi->mi_lock);
488                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
489                         mutex_exit(&mi->mi_lock);
490                 }

492                 do {
493                         tsize = MIN(mi->mi_tsize, count);
494                         args.offset = (offset3)offset;
495                         args.count = (count3)tsize;
496                         res.size = (uint_t)tsize;
497                         args.res_uiop = uiop;
498                         args.res_data_val_alt = NULL;

500                         error = rfs3call(mi, NFSPROC3_READ,
501                             xdr_READ3args, (caddr_t)&args,
502                             xdr_READ3uiores, (caddr_t)&res, cr,
503                             &douprintf, &res.status, 0, &fi);
504                 } while (error == ENFS_TRYAGAIN);

506                 if (mi->mi_io_kstats) {
507                         mutex_enter(&mi->mi_lock);
508                         kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
509                         mutex_exit(&mi->mi_lock);
510                 }

512                 if (error)
513                         return (error);

515                 error = geterrno3(res.status);
516                 if (error)
517                         return (error);

519                 if (res.count != res.size) {
520                         zcmn_err(getzoneid(), CE_WARN,
521 "nfs3_directio_read: server %s returned incorrect amount",
522                             sv_hostname);
523                         return (EIO);
```

```
524                 }
525                 count -= res.count;
526                 offset += res.count;
527                 if (mi->mi_io_kstats) {
528                         mutex_enter(&mi->mi_lock);
529                         KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
530                         KSTAT_IO_PTR(mi->mi_io_kstats)->nread += res.count;
531                         mutex_exit(&mi->mi_lock);
532                 }
533                 lwp_stat_update(LWP_STAT_INBLK, 1);
534         } while (count && !res.eof);

536         return (0);
537 }

539 /* ARGSUSED */
540 static int
541 nfs3_read(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
542         caller_context_t *ct)
543 {
544         rnode_t *rp;
545         u_offset_t off;
546         offset_t diff;
547         int on;
548         size_t n;
549         caddr_t base;
550         uint_t flags;
551         int error = 0;
552         mntinfo_t *mi;

554         rp = VTOR(vp);
555         mi = VTOMI(vp);

557         ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));

559         if (nfs_zone() != mi->mi_zone)
560                 return (EIO);

562         if (vp->v_type != VREG)
563                 return (EISDIR);

565         if (uiop->uio_resid == 0)
566                 return (0);

568         if (uiop->uio_loffset < 0 || uiop->uio_loffset + uiop->uio_resid < 0)
569                 return (EINVAL);

571         /*
572          * Bypass VM if caching has been disabled (e.g., locking) or if
573          * using client-side direct I/O and the file is not mmap'd and
574          * there are no cached pages.
575          */
576         if ((vp->v_flag & VNOCACHE) ||
577             (((rp->r_flags & RDIRECTIO) || (mi->mi_flags & MI_DIRECTIO)) &&
578             rp->r_mapcnt == 0 && rp->r_inmap == 0 &&
579             !vn_has_cached_data(vp))) {
580                 return (nfs3_directio_read(vp, uiop, cr));
581         }

583         do {
584                 off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
585                 on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
586                 n = MIN(MAXBSIZE - on, uiop->uio_resid);

588                 error = nfs3_validate_caches(vp, cr);
589                 if (error)
```



```
590                         break;

592                 mutex_enter(&rp->r_statelock);
593                 while (rp->r_flags & RINCACHEPURGE) {
594                         if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
595                                 mutex_exit(&rp->r_statelock);
596                                 return (EINTR);
597                         }
598                 }
599                 diff = rp->r_size - uiop->uio_loffset;
600                 mutex_exit(&rp->r_statelock);
601                 if (diff <= 0)
602                         break;
603                 if (diff < n)
604                         n = (size_t)diff;

606                 if (vpm_enable) {
607                         /*
608                          * Copy data.
609                          */
610                         error = vpm_data_copy(vp, off + on, n, uiop,
611                             1, NULL, 0, S_READ);
612                 } else {
613                         base = segmap_getmapflt(segkmap, vp, off + on, n, 1,
614                             S_READ);

616                         error = uiomove(base + on, n, UIO_READ, uiop);
617                 }

619                 if (!error) {
620                         /*
621                          * If read a whole block or read to eof,
622                          * won't need this buffer again soon.
623                          */
624                         mutex_enter(&rp->r_statelock);
625                         if (n + on == MAXBSIZE ||
626                             uiop->uio_loffset == rp->r_size)
627                                 flags = SM_DONTNEED;
628                         else
629                                 flags = 0;
630                         mutex_exit(&rp->r_statelock);
631                         if (vpm_enable) {
632                                 error = vpm_sync_pages(vp, off, n, flags);
633                         } else {
634                                 error = segmap_release(segkmap, base, flags);
635                         }
636                 } else {
637                         if (vpm_enable) {
638                                 (void) vpm_sync_pages(vp, off, n, 0);
639                         } else {
640                                 (void) segmap_release(segkmap, base, 0);
641                         }
642                 }
643         } while (!error && uiop->uio_resid > 0);

645         return (error);
646 }

648 /* ARGSUSED */
649 static int
650 nfs3_write(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
651         caller_context_t *ct)
652 {
653         rlim64_t limit = uiop->uio_llimit;
654         rnode_t *rp;
655         u_offset_t off;
```

```
656         caddr_t base;
657         uint_t flags;
658         int remainder;
659         size_t n;
660         int on;
661         int error;
662         int resid;
663         offset_t offset;
664         mntinfo_t *mi;
665         uint_t bsize;

667         rp = VTOR(vp);

669         if (vp->v_type != VREG)
670                 return (EISDIR);

672         mi = VTOMI(vp);
673         if (nfs_zone() != mi->mi_zone)
674                 return (EIO);
675         if (uiop->uio_resid == 0)
676                 return (0);

678         if (ioflag & FAPPEND) {
679                 struct vattr va;

681                 /*
682                  * Must serialize if appending.
683                  */
684                 if (nfs_rw_lock_held(&rp->r_rwlock, RW_READER)) {
685                         nfs_rw_exit(&rp->r_rwlock);
686                         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER,
687                             INTR(vp)))
688                                 return (EINTR);
689                 }

691                 va.va_mask = AT_SIZE;
692                 error = nfs3getattr(vp, &va, cr);
693                 if (error)
694                         return (error);
695                 uiop->uio_loffset = va.va_size;
696         }

698         offset = uiop->uio_loffset + uiop->uio_resid;

700         if (uiop->uio_loffset < 0 || offset < 0)
701                 return (EINVAL);

703         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
704                 limit = MAXOFFSET_T;

706         /*
707          * Check to make sure that the process will not exceed
708          * its limit on file size.  It is okay to write up to
709          * the limit, but not beyond.  Thus, the write which
710          * reaches the limit will be short and the next write
711          * will return an error.
712          */
713         remainder = 0;
714         if (offset > limit) {
715                 remainder = offset - limit;
716                 uiop->uio_resid = limit - uiop->uio_loffset;
717                 if (uiop->uio_resid <= 0) {
718                         proc_t *p = ttoproc(curthread);

720                         uiop->uio_resid += remainder;
721                         mutex_enter(&p->p_lock);
```



```
722                         (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE],
723                             p->p_rctls, p, RCA_UNSAFE_SIGINFO);
724                         mutex_exit(&p->p_lock);
725                         return (EFBIG);
726                 }
727         }

729         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR(vp)))
730                 return (EINTR);

732         /*
733          * Bypass VM if caching has been disabled (e.g., locking) or if
734          * using client-side direct I/O and the file is not mmap'd and
735          * there are no cached pages.
736          */
737         if ((vp->v_flag & VNOCACHE) ||
738             (((rp->r_flags & RDIRECTIO) || (mi->mi_flags & MI_DIRECTIO)) &&
739             rp->r_mapcnt == 0 && rp->r_inmap == 0 &&
740             !vn_has_cached_data(vp))) {
741                 size_t bufsize;
742                 int count;
743                 u_offset_t org_offset;
744                 stable_how stab_comm;

746 nfs3_fwrite:
747                 if (rp->r_flags & RSTALE) {
748                         resid = uiop->uio_resid;
749                         offset = uiop->uio_loffset;
750                         error = rp->r_error;
751                         /*
752                          * A close may have cleared r_error, if so,
753                          * propagate ESTALE error return properly
754                          */
755                         if (error == 0)
756                                 error = ESTALE;
757                         goto bottom;
758                 }
759                 bufsize = MIN(uiop->uio_resid, mi->mi_stsize);
760                 base = kmem_alloc(bufsize, KM_SLEEP);
761                 do {
762                         if (ioflag & FDSYNC)
763                                 stab_comm = DATA_SYNC;
764                         else
765                                 stab_comm = FILE_SYNC;
766                         resid = uiop->uio_resid;
767                         offset = uiop->uio_loffset;
768                         count = MIN(uiop->uio_resid, bufsize);
769                         org_offset = uiop->uio_loffset;
770                         error = uiomove(base, count, UIO_WRITE, uiop);
771                         if (!error) {
772                                 error = nfs3write(vp, base, org_offset,
773                                     count, cr, &stab_comm);
774                         }
775                 } while (!error && uiop->uio_resid > 0);
776                 kmem_free(base, bufsize);
777                 goto bottom;
778         }


781         bsize = vp->v_vfsp->vfs_bsize;

783         do {
784                 off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
785                 on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
786                 n = MIN(MAXBSIZE - on, uiop->uio_resid);
```

```
788                 resid = uiop->uio_resid;
789                 offset = uiop->uio_loffset;
790
791                 if (rp->r_flags & RSTALE) {
792                         error = rp->r_error;
793                         /*
794                          * A close may have cleared r_error, if so,
795                          * propagate ESTALE error return properly
796                          */
797                         if (error == 0)
798                                 error = ESTALE;
799                         break;
800                 }
801
802                 /*
803                  * Don't create dirty pages faster than they
804                  * can be cleaned so that the system doesn't
805                  * get imbalanced.  If the async queue is
806                  * maxed out, then wait for it to drain before
807                  * creating more dirty pages.  Also, wait for
808                  * any threads doing pagewalks in the vop_getattr
809                  * entry points so that they don't block for
810                  * long periods.
811                  */
812                 mutex_enter(&rp->r_statelock);
813                 while ((mi->mi_max_threads != 0 &&
814                     rp->r_awcount > 2 * mi->mi_max_threads) ||
815                     rp->r_gcount > 0) {
816                         if (INTR(vp)) {
817                                 klwp_t *lwp = ttolwp(curthread);
818
819                                 if (lwp != NULL)
820                                         lwp->lwp_nostop++;
821                                 if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
822                                         mutex_exit(&rp->r_statelock);
823                                         if (lwp != NULL)
824                                                 lwp->lwp_nostop--;
825                                         error = EINTR;
826                                         goto bottom;
827                                 }
828                                 if (lwp != NULL)
829                                         lwp->lwp_nostop--;
830                         } else
831                                 cv_wait(&rp->r_cv, &rp->r_statelock);
832                 }
833                 mutex_exit(&rp->r_statelock);
834
835                 /*
836                  * Touch the page and fault it in if it is not in core
837                  * before segmap_getmapflt or vpm_data_copy can lock it.
838                  * This is to avoid the deadlock if the buffer is mapped
839                  * to the same file through mmap which we want to write.
840                  */
841                 uio_prefaultpages((long)n, uiop);
842
843                 if (vpm_enable) {
844                         /*
845                          * It will use kpm mappings, so no need to
846                          * pass an address.
847                          */
848                         error = writerp(rp, NULL, n, uiop, 0);
849                 } else  {
850                         if (segmap_kpm) {
851                                 int pon = uiop->uio_loffset & PAGEOFFSET;
852                                 size_t pn = MIN(PAGESIZE - pon,
853                                     uiop->uio_resid);
```



```
854                                 int pagecreate;
855
856                                 mutex_enter(&rp->r_statelock);
857                                 pagecreate = (pon == 0) && (pn == PAGESIZE ||
858                                     uiop->uio_loffset + pn >= rp->r_size);
859                                 mutex_exit(&rp->r_statelock);
860
861                                 base = segmap_getmapflt(segkmap, vp, off + on,
862                                     pn, !pagecreate, S_WRITE);
863
864                                 error = writerp(rp, base + pon, n, uiop,
865                                     pagecreate);
866
867                         } else {
868                                 base = segmap_getmapflt(segkmap, vp, off + on,
869                                     n, 0, S_READ);
870                                 error = writerp(rp, base + on, n, uiop, 0);
871                         }
872                 }
873
874                 if (!error) {
875                         if (mi->mi_flags & MI_NOAC)
876                                 flags = SM_WRITE;
877                         else if ((uiop->uio_loffset % bsize) == 0 ||
878                             IS_SWAPVP(vp)) {
879                                 /*
880                                  * Have written a whole block.
881                                  * Start an asynchronous write
882                                  * and mark the buffer to
883                                  * indicate that it won't be
884                                  * needed again soon.
885                                  */
886                                 flags = SM_WRITE | SM_ASYNC | SM_DONTNEED;
887                         } else
888                                 flags = 0;
889                         if ((ioflag & (FSYNC|FDSYNC)) ||
890                             (rp->r_flags & ROUTOFSPACE)) {
891                                 flags &= ~SM_ASYNC;
892                                 flags |= SM_WRITE;
893                         }
894                         if (vpm_enable) {
895                                 error = vpm_sync_pages(vp, off, n, flags);
896                         } else {
897                                 error = segmap_release(segkmap, base, flags);
898                         }
899                 } else {
900                         if (vpm_enable) {
901                                 (void) vpm_sync_pages(vp, off, n, 0);
902                         } else {
903                                 (void) segmap_release(segkmap, base, 0);
904                         }
905                         /*
906                          * In the event that we got an access error while
907                          * faulting in a page for a write-only file just
908                          * force a write.
909                          */
910                         if (error == EACCES)
911                                 goto nfs3_fwrite;
912                 }
913         } while (!error && uiop->uio_resid > 0);
914
915 bottom:
916         if (error) {
917                 uiop->uio_resid = resid + remainder;
918                 uiop->uio_loffset = offset;
919         } else
```

```
920                 uiop->uio_resid += remainder;

922         nfs_rw_exit(&rp->r_lkserlock);

924         return (error);
925 }

927 /*
928  * Flags are composed of {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED}
929  */
930 static int
931 nfs3_rdwrlbn(vnode_t *vp, page_t *pp, u_offset_t off, size_t len,
932         int flags, cred_t *cr)
933 {
934         struct buf *bp;
935         int error;
936         page_t *savepp;
937         uchar_t fsdata;
938         stable_how stab_comm;

940         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);
941         bp = pageio_setup(pp, len, vp, flags);
942         ASSERT(bp != NULL);

944         /*
945          * pageio_setup should have set b_addr to 0.  This
946          * is correct since we want to do I/O on a page
947          * boundary.  bp_mapin will use this addr to calculate
948          * an offset, and then set b_addr to the kernel virtual
949          * address it allocated for us.
950          */
951         ASSERT(bp->b_un.b_addr == 0);

953         bp->b_edev = 0;
954         bp->b_dev = 0;
955         bp->b_lblkno = lbtodb(off);
956         bp->b_file = vp;
957         bp->b_offset = (offset_t)off;
958         bp_mapin(bp);

960         /*
961          * Calculate the desired level of stability to write data
962          * on the server and then mark all of the pages to reflect
963          * this.
964          */
965         if ((flags & (B_WRITE|B_ASYNC)) == (B_WRITE|B_ASYNC) &&
966             freemem > desfree) {
967                 stab_comm = UNSTABLE;
968                 fsdata = C_DELAYCOMMIT;
969         } else {
970                 stab_comm = FILE_SYNC;
971                 fsdata = C_NOCOMMIT;
972         }

974         savepp = pp;
975         do {
976                 pp->p_fsdata = fsdata;
977         } while ((pp = pp->p_next) != savepp);

979         error = nfs3_bio(bp, &stab_comm, cr);

981         bp_mapout(bp);
982         pageio_done(bp);

984         /*
985          * If the server wrote pages in a more stable fashion than
```



```
986          * was requested, then clear all of the marks in the pages
987          * indicating that COMMIT operations were required.
988          */
989         if (stab_comm != UNSTABLE && fsdata == C_DELAYCOMMIT) {
990                 do {
991                         pp->p_fsdata = C_NOCOMMIT;
992                 } while ((pp = pp->p_next) != savepp);
993         }

995         return (error);
996 }

998 /*
999  * Write to file.  Writes to remote server in largest size
1000  * chunks that the server can handle.  Write is synchronous.
1001  */
1002 static int
1003 nfs3write(vnode_t *vp, caddr_t base, u_offset_t offset, int count, cred_t *cr,
1004         stable_how *stab_comm)
1005 {
1006         mntinfo_t *mi;
1007         WRITE3args args;
1008         WRITE3res res;
1009         int error;
1010         int tsize;
1011         rnode_t *rp;
1012         int douprintf;

1014         rp = VTOR(vp);
1015         mi = VTOMI(vp);

1017         ASSERT(nfs_zone() == mi->mi_zone);

1019         args.file = *VTOFH3(vp);
1020         args.stable = *stab_comm;

1022         *stab_comm = FILE_SYNC;

1024         douprintf = 1;

1026         do {
1027                 if ((vp->v_flag & VNOCACHE) ||
1028                     (rp->r_flags & RDIRECTIO) ||
1029                     (mi->mi_flags & MI_DIRECTIO))
1030                         tsize = MIN(mi->mi_stsize, count);
1031                 else
1032                         tsize = MIN(mi->mi_curwrite, count);
1033                 args.offset = (offset3)offset;
1034                 args.count = (count3)tsize;
1035                 args.data.data_len = (uint_t)tsize;
1036                 args.data.data_val = base;

1038                 if (mi->mi_io_kstats) {
1039                         mutex_enter(&mi->mi_lock);
1040                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
1041                         mutex_exit(&mi->mi_lock);
1042                 }
1043                 args.mblk = NULL;
1044                 do {
1045                         error = rfs3call(mi, NFSPROC3_WRITE,
1046                             xdr_WRITE3args, (caddr_t)&args,
1047                             xdr_WRITE3res, (caddr_t)&res, cr,
1048                             &douprintf, &res.status, 0, NULL);
1049                 } while (error == ENFS_TRYAGAIN);
1050                 if (mi->mi_io_kstats) {
1051                         mutex_enter(&mi->mi_lock);
```

```
1052                    kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
1053                    mutex_exit(&mi->mi_lock);
1054            }

1056            if (error)
1057                    return (error);
1058            error = geterrno3(res.status);
1059            if (!error) {
1060                    if (res.resok.count > args.count) {
1061                            zcmn_err(getzoneid(), CE_WARN,
1062                                "nfs3write: server %s wrote %u, "
1063                                "requested was %u",
1064                                rp->r_server->sv_hostname,
1065                                res.resok.count, args.count);
1066                            return (EIO);
1067                    }
1068                    if (res.resok.committed == UNSTABLE) {
1069                            *stab_comm = UNSTABLE;
1070                            if (args.stable == DATA_SYNC ||
1071                                args.stable == FILE_SYNC) {
1072                                    zcmn_err(getzoneid(), CE_WARN,
1073                    "nfs3write: server %s did not commit to stable storage",
1074                                        rp->r_server->sv_hostname);
1075                                return (EIO);
1076                            }
1077                    }
1078                    tsize = (int)res.resok.count;
1079                    count -= tsize;
1080                    base += tsize;
1081                    offset += tsize;
1082                    if (mi->mi_io_kstats) {
1083                            mutex_enter(&mi->mi_lock);
1084                            KSTAT_IO_PTR(mi->mi_io_kstats)->writes++;
1085                            KSTAT_IO_PTR(mi->mi_io_kstats)->nwritten +=
1086                                tsize;
1087                            mutex_exit(&mi->mi_lock);
1088                    }
1089                    lwp_stat_update(LWP_STAT_OUBLK, 1);
1090                    mutex_enter(&rp->r_statelock);
1091                    if (rp->r_flags & RHAVEVERF) {
1092                            if (rp->r_verf != res.resok.verf) {
1093                                    nfs3_set_mod(vp);
1094                                    rp->r_verf = res.resok.verf;
1095                                    /*
1096                                     * If the data was written UNSTABLE,
1097                                     * then might as well stop because
1098                                     * the whole block will have to get
1099                                     * rewritten anyway.
1100                                     */
1101                                    if (*stab_comm == UNSTABLE) {
1102                                            mutex_exit(&rp->r_statelock);
1103                                            break;
1104                                    }
1105                            }
1106                    } else {
1107                            rp->r_verf = res.resok.verf;
1108                            rp->r_flags |= RHAVEVERF;
1109                    }
1110                    /*
1111                     * Mark the attribute cache as timed out and
1112                     * set RWRITEATTR to indicate that the file
1113                     * was modified with a WRITE operation and
1114                     * that the attributes can not be trusted.
1115                     */
1116                    PURGE_ATTRCACHE_LOCKED(rp);
1117                    rp->r_flags |= RWRITEATTR;
```



```
1118                    mutex_exit(&rp->r_statelock);
1119                }
1120            } while (!error && count);

1122            return (error);
1123    }

1125    /*
1126     * Read from a file.  Reads data in largest chunks our interface can handle.
1127     */
1128    static int
1129    nfs3read(vnode_t *vp, caddr_t base, offset_t offset, int count,
1130            size_t *residp, cred_t *cr)
1131    {
1132            mntinfo_t *mi;
1133            READ3args args;
1134            READ3vres res;
1135            int tsize;
1136            int error;
1137            int douprintf;
1138            failinfo_t fi;
1139            rnode_t *rp;
1140            struct vattr va;
1141            hrtime_t t;

1143            rp = VTOR(vp);
1144            mi = VTOMI(vp);
1145            ASSERT(nfs_zone() == mi->mi_zone);
1146            douprintf = 1;

1148            args.file = *VTOFH3(vp);
1149            fi.vp = vp;
1150            fi.fhp = (caddr_t)&args.file;
1151            fi.copyproc = nfs3copyfh;
1152            fi.lookupproc = nfs3lookup;
1153            fi.xattrdirproc = acl_getxattrdir3;

1155            res.pov.fres.vp = vp;
1156            res.pov.fres.vap = &va;

1158            res.wlist = NULL;
1159            *residp = count;
1160            do {
1161                    if (mi->mi_io_kstats) {
1162                            mutex_enter(&mi->mi_lock);
1163                            kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
1164                            mutex_exit(&mi->mi_lock);
1165                    }

1167                    do {
1168                            if ((vp->v_flag & VNOCACHE) ||
1169                                (rp->r_flags & RDIRECTIO) ||
1170                                (mi->mi_flags & MI_DIRECTIO))
1171                                    tsize = MIN(mi->mi_tsize, count);
1172                            else
1173                                    tsize = MIN(mi->mi_curread, count);
1174                            res.data.data_val = base;
1175                            res.data.data_len = tsize;
1176                            args.offset = (offset3)offset;
1177                            args.count = (count3)tsize;
1178                            args.res_uiop = NULL;
1179                            args.res_data_val_alt = base;

1181                            t = gethrtime();
1182                            error = rfs3call(mi, NFSPROC3_READ,
1183                                xdr_READ3args, (caddr_t)&args,
```

```
1184                            xdr_READ3vres, (caddr_t)&res, cr,
1185                            &douprintf, &res.status, 0, &fi);
1186                } while (error == ENFS_TRYAGAIN);

1188                if (mi->mi_io_kstats) {
1189                        mutex_enter(&mi->mi_lock);
1190                        kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
1191                        mutex_exit(&mi->mi_lock);
1192                }

1194                if (error)
1195                        return (error);

1197                error = geterrno3(res.status);
1198                if (error)
1199                        return (error);

1201                if (res.count != res.data.data_len) {
1202                        zcmn_err(getzoneid(), CE_WARN,
1203                            "nfs3read: server %s returned incorrect amount",
1204                            rp->r_server->sv_hostname);
1205                        return (EIO);
1206                }

1208                count -= res.count;
1209                *residp = count;
1210                base += res.count;
1211                offset += res.count;
1212                if (mi->mi_io_kstats) {
1213                        mutex_enter(&mi->mi_lock);
1214                        KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
1215                        KSTAT_IO_PTR(mi->mi_io_kstats)->nread += res.count;
1216                        mutex_exit(&mi->mi_lock);
1217                }
1218                lwp_stat_update(LWP_STAT_INBLK, 1);
1219        } while (count && !res.eof);

1221        if (res.pov.attributes) {
1222                mutex_enter(&rp->r_statelock);
1223                if (!CACHE_VALID(rp, va.va_mtime, va.va_size)) {
1224                        mutex_exit(&rp->r_statelock);
1225                        PURGE_ATTRCACHE(vp);
1226                } else {
1227                        if (rp->r_mtime <= t)
1228                                nfs_attrcache_va(vp, &va);
1229                        mutex_exit(&rp->r_statelock);
1230                }
1231        }

1233        return (0);
1234 }

1236 /* ARGSUSED */
1237 static int
1238 nfs3_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp,
1239         caller_context_t *ct)
1240 {

1242        if (nfs_zone() != VTOMI(vp)->mi_zone)
1243                return (EIO);
1244        switch (cmd) {
1245                case _FIODIRECTIO:
1246                        return (nfs_directio(vp, (int)arg, cr));
1247                default:
1248                        return (ENOTTY);
1249        }
```



```
1250 }

1252 /* ARGSUSED */
1253 static int
1254 nfs3_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1255         caller_context_t *ct)
1256 {
1257        int error;
1258        rnode_t *rp;

1260        if (nfs_zone() != VTOMI(vp)->mi_zone)
1261                return (EIO);
1262        /*
1263         * If it has been specified that the return value will
1264         * just be used as a hint, and we are only being asked
1265         * for size, fsid or rdevid, then return the client's
1266         * notion of these values without checking to make sure
1267         * that the attribute cache is up to date.
1268         * The whole point is to avoid an over the wire GETATTR
1269         * call.
1270         */
1271        rp = VTOR(vp);
1272        if (flags & ATTR_HINT) {
1273                if (vap->va_mask ==
1274                    (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
1275                        mutex_enter(&rp->r_statelock);
1276                        if (vap->va_mask | AT_SIZE)
1277                                vap->va_size = rp->r_size;
1278                        if (vap->va_mask | AT_FSID)
1279                                vap->va_fsid = rp->r_attr.va_fsid;
1280                        if (vap->va_mask | AT_RDEV)
1281                                vap->va_rdev = rp->r_attr.va_rdev;
1282                        mutex_exit(&rp->r_statelock);
1283                        return (0);
1284                }
1285        }

1287        /*
1288         * Only need to flush pages if asking for the mtime
1289         * and if there any dirty pages or any outstanding
1290         * asynchronous (write) requests for this file.
1291         */
1292        if (vap->va_mask & AT_MTIME) {
1293                if (vn_has_cached_data(vp) &&
1294                    ((rp->r_flags & RDIRTY) || rp->r_awcount > 0)) {
1295                        mutex_enter(&rp->r_statelock);
1296                        rp->r_gcount++;
1297                        mutex_exit(&rp->r_statelock);
1298                        error = nfs3_putpage(vp, (offset_t)0, 0, 0, cr, ct);
1299                        mutex_enter(&rp->r_statelock);
1300                        if (error && (error == ENOSPC || error == EDQUOT)) {
1301                                if (!rp->r_error)
1302                                        rp->r_error = error;
1303                        }
1304                        if (--rp->r_gcount == 0)
1305                                cv_broadcast(&rp->r_cv);
1306                        mutex_exit(&rp->r_statelock);
1307                }
1308        }

1310        return (nfs3getattr(vp, vap, cr));
1311 }

1313 /*ARGSUSED4*/
1314 static int
1315 nfs3_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
```

```
1316                 caller_context_t *ct)
1317 {
1318         int error;
1319         struct vattr va;

1321         if (vap->va_mask & AT_NOSET)
1322                 return (EINVAL);
1323         if (nfs_zone() != VTOMI(vp)->mi_zone)
1324                 return (EIO);

1326         va.va_mask = AT_UID | AT_MODE;
1327         error = nfs3getattr(vp, &va, cr);
1328         if (error)
1329                 return (error);

1331         error = secpolicy_vnode_setattr(cr, vp, vap, &va, flags, nfs3_accessx,
1332             vp);
1333         if (error)
1334                 return (error);

1336         error = nfs3setattr(vp, vap, flags, cr);

1338         if (error == 0 && (vap->va_mask & AT_SIZE) && vap->va_size == 0)
1339                 vnevent_truncate(vp, ct);

1341         return (error);
1342 }

1344 static int
1345 nfs3setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr)
1346 {
1347         int error;
1348         uint_t mask;
1349         SETATTR3args args;
1350         SETATTR3res res;
1351         int douprintf;
1352         rnode_t *rp;
1353         struct vattr va;
1354         mode_t omode;
1355         vsecattr_t *vsp;
1356         hrtime_t t;

1358         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);
1359         mask = vap->va_mask;

1361         rp = VTOR(vp);

1363         /*
1364          * Only need to flush pages if there are any pages and
1365          * if the file is marked as dirty in some fashion.  The
1366          * file must be flushed so that we can accurately
1367          * determine the size of the file and the cached data
1368          * after the SETATTR returns.  A file is considered to
1369          * be dirty if it is either marked with RDIRTY, has
1370          * outstanding i/o's active, or is mmap'd.  In this
1371          * last case, we can't tell whether there are dirty
1372          * pages, so we flush just to be sure.
1373          */
1374         if (vn_has_cached_data(vp) &&
1375             ((rp->r_flags & RDIRTY) ||
1376             rp->r_count > 0 ||
1377             rp->r_mapcnt > 0)) {
1378                 ASSERT(vp->v_type != VCHR);
1379                 error = nfs3_putpage(vp, (offset_t)0, 0, 0, cr, NULL);
1380                 if (error && (error == ENOSPC || error == EDQUOT)) {
1381                         mutex_enter(&rp->r_statelock);
```



```
1382                         if (!rp->r_error)
1383                                 rp->r_error = error;
1384                         mutex_exit(&rp->r_statelock);
1385                 }
1386         }

1388         args.object = *RTOFH3(rp);
1389         /*
1390          * If the intent is for the server to set the times,
1391          * there is no point in have the mask indicating set mtime or
1392          * atime, because the vap values may be junk, and so result
1393          * in an overflow error. Remove these flags from the vap mask
1394          * before calling in this case, and restore them afterwards.
1395          */
1396         if ((mask & (AT_ATIME | AT_MTIME)) && !(flags & ATTR_UTIME)) {
1397                 /* Use server times, so don't set the args time fields */
1398                 vap->va_mask &= ~(AT_ATIME | AT_MTIME);
1399                 error = vattr_to_sattr3(vap, &args.new_attributes);
1400                 vap->va_mask |= (mask & (AT_ATIME | AT_MTIME));
1401                 if (mask & AT_ATIME) {
1402                         args.new_attributes.atime.set_it = SET_TO_SERVER_TIME;
1403                 }
1404                 if (mask & AT_MTIME) {
1405                         args.new_attributes.mtime.set_it = SET_TO_SERVER_TIME;
1406                 }
1407         } else {
1408                 /* Either do not set times or use the client specified times */
1409                 error = vattr_to_sattr3(vap, &args.new_attributes);
1410         }

1412         if (error) {
1413                 /* req time field(s) overflow - return immediately */
1414                 return (error);
1415         }

1417         va.va_mask = AT_MODE | AT_CTIME;
1418         error = nfs3getattr(vp, &va, cr);
1419         if (error)
1420                 return (error);
1421         omode = va.va_mode;

1423 tryagain:
1424         if (mask & AT_SIZE) {
1425                 args.guard.check = TRUE;
1426                 args.guard.obj_ctime.seconds = va.va_ctime.tv_sec;
1427                 args.guard.obj_ctime.nseconds = va.va_ctime.tv_nsec;
1428         } else
1429                 args.guard.check = FALSE;

1431         douprintf = 1;

1433         t = gethrtime();

1435         error = rfs3call(VTOMI(vp), NFSPROC3_SETATTR,
1436             xdr_SETATTR3args, (caddr_t)&args,
1437             xdr_SETATTR3res, (caddr_t)&res, cr,
1438             &douprintf, &res.status, 0, NULL);

1440         /*
1441          * Purge the access cache and ACL cache if changing either the
1442          * owner of the file, the group owner, or the mode.  These may
1443          * change the access permissions of the file, so purge old
1444          * information and start over again.
1445          */
1446         if (mask & (AT_UID | AT_GID | AT_MODE)) {
1447                 (void) nfs_access_purge_rp(rp);
```

```
1448                if (rp->r_secattr != NULL) {
1449                        mutex_enter(&rp->r_statelock);
1450                        vsp = rp->r_secattr;
1451                        rp->r_secattr = NULL;
1452                        mutex_exit(&rp->r_statelock);
1453                        if (vsp != NULL)
1454                                nfs_acl_free(vsp);
1455                }
1456        }

1458        if (error) {
1459                PURGE_ATTRCACHE(vp);
1460                return (error);
1461        }

1463        error = geterrno3(res.status);
1464        if (!error) {
1465                /*
1466                 * If changing the size of the file, invalidate
1467                 * any local cached data which is no longer part
1468                 * of the file.  We also possibly invalidate the
1469                 * last page in the file.  We could use
1470                 * pvn_vpzero(), but this would mark the page as
1471                 * modified and require it to be written back to
1472                 * the server for no particularly good reason.
1473                 * This way, if we access it, then we bring it
1474                 * back in.  A read should be cheaper than a
1475                 * write.
1476                 */
1477                if (mask & AT_SIZE) {
1478                        nfs_invalidate_pages(vp,
1479                            (vap->va_size & PAGEMASK), cr);
1480                }
1481                nfs3_cache_wcc_data(vp, &res.resok.obj_wcc, t, cr);
1482                /*
1483                 * Some servers will change the mode to clear the setuid
1484                 * and setgid bits when changing the uid or gid.  The
1485                 * client needs to compensate appropriately.
1486                 */
1487                if (mask & (AT_UID | AT_GID)) {
1488                        int terror;

1490                        va.va_mask = AT_MODE;
1491                        terror = nfs3getattr(vp, &va, cr);
1492                        if (!terror &&
1493                            (((mask & AT_MODE) && va.va_mode != vap->va_mode) ||
1494                            (!(mask & AT_MODE) && va.va_mode != omode))) {
1495                                va.va_mask = AT_MODE;
1496                                if (mask & AT_MODE)
1497                                        va.va_mode = vap->va_mode;
1498                                else
1499                                        va.va_mode = omode;
1500                                (void) nfs3setattr(vp, &va, 0, cr);
1501                        }
1502                }
1503        } else {
1504                nfs3_cache_wcc_data(vp, &res.resfail.obj_wcc, t, cr);
1505                /*
1506                 * If we got back a "not synchronized" error, then
1507                 * we need to retry with a new guard value.  The
1508                 * guard value used is the change time.  If the
1509                 * server returned post_op_attr, then we can just
1510                 * retry because we have the latest attributes.
1511                 * Otherwise, we issue a GETATTR to get the latest
1512                 * attributes and then retry.  If we couldn't get
1513                 * the attributes this way either, then we give
```



```
1514                 * up because we can't complete the operation as
1515                 * required.
1516                 */
1517                if (res.status == NFS3ERR_NOT_SYNC) {
1518                        va.va_mask = AT_CTIME;
1519                        if (nfs3getattr(vp, &va, cr) == 0)
1520                                goto tryagain;
1521                }
1522                PURGE_STALE_FH(error, vp, cr);
1523        }

1525        return (error);
1526 }

1528 static int
1529 nfs3_accessx(void *vp, int mode, cred_t *cr)
1530 {
1531        ASSERT(nfs_zone() == VTOMI((vnode_t *)vp)->mi_zone);
1532        return (nfs3_access(vp, mode, 0, cr, NULL));
1533 }

1535 /* ARGSUSED */
1536 static int
1537 nfs3_access(vnode_t *vp, int mode, int flags, cred_t *cr, caller_context_t *ct)
1538 {
1539        int error;
1540        ACCESS3args args;
1541        ACCESS3res res;
1542        int douprintf;
1543        uint32 acc;
1544        rnode_t *rp;
1545        cred_t *cred, *ncr, *ncrfree = NULL;
1546        failinfo_t fi;
1547        nfs_access_type_t cacc;
1548        hrtime_t t;

1550        acc = 0;
1551        if (nfs_zone() != VTOMI(vp)->mi_zone)
1552                return (EIO);
1553        if (mode & VREAD)
1554                acc |= ACCESS3_READ;
1555        if (mode & VWRITE) {
1556                if (vn_is_readonly(vp) && !IS_DEVVP(vp))
1557                        return (EROFS);
1558                if (vp->v_type == VDIR)
1559                        acc |= ACCESS3_DELETE;
1560                acc |= ACCESS3_MODIFY | ACCESS3_EXTEND;
1561        }
1562        if (mode & VEXEC) {
1563                if (vp->v_type == VDIR)
1564                        acc |= ACCESS3_LOOKUP;
1565                else
1566                        acc |= ACCESS3_EXECUTE;
1567        }

1569        rp = VTOR(vp);
1570        args.object = *VTOFH3(vp);
1571        if (vp->v_type == VDIR) {
1572                args.access = ACCESS3_READ | ACCESS3_DELETE | ACCESS3_MODIFY |
1573                    ACCESS3_EXTEND | ACCESS3_LOOKUP;
1574        } else {
1575                args.access = ACCESS3_READ | ACCESS3_MODIFY | ACCESS3_EXTEND |
1576                    ACCESS3_EXECUTE;
1577        }
1578        fi.vp = vp;
1579        fi.fhp = (caddr_t)&args.object;
```

```
1580         fi.copyproc = nfs3copyfh;
1581         fi.lookupproc = nfs3lookup;
1582         fi.xattrdirproc = acl_getxattrdir3;

1584         cred = cr;
1585         /*
1586          * ncr and ncrfree both initially
1587          * point to the memory area returned
1588          * by crnetadjust();
1589          * ncrfree not NULL when exiting means
1590          * that we need to release it
1591          */
1592         ncr = crnetadjust(cred);
1593         ncrfree = ncr;
1594 tryagain:
1595         if (rp->r_acache != NULL) {
1596                 cacc = nfs_access_check(rp, acc, cred);
1597                 if (cacc == NFS_ACCESS_ALLOWED) {
1598                         if (ncrfree != NULL)
1599                                 crfree(ncrfree);
1600                         return (0);
1601                 }
1602                 if (cacc == NFS_ACCESS_DENIED) {
1603                         /*
1604                          * If the cred can be adjusted, try again
1605                          * with the new cred.
1606                          */
1607                         if (ncr != NULL) {
1608                                 cred = ncr;
1609                                 ncr = NULL;
1610                                 goto tryagain;
1611                         }
1612                         if (ncrfree != NULL)
1613                                 crfree(ncrfree);
1614                         return (EACCES);
1615                 }
1616         }

1618         douprintf = 1;

1620         t = gethrtime();

1622         error = rfs3call(VTOMI(vp), NFSPROC3_ACCESS,
1623             xdr_ACCESS3args, (caddr_t)&args,
1624             xdr_ACCESS3res, (caddr_t)&res, cred,
1625             &douprintf, &res.status, 0, &fi);

1627         if (error) {
1628                 if (ncrfree != NULL)
1629                         crfree(ncrfree);
1630                 return (error);
1631         }

1633         error = geterrno3(res.status);
1634         if (!error) {
1635                 nfs3_cache_post_op_attr(vp, &res.resok.obj_attributes, t, cr);
1636                 nfs_access_cache(rp, args.access, res.resok.access, cred);
1637                 /*
1638                  * we just cached results with cred; if cred is the
1639                  * adjusted credentials from crnetadjust, we do not want
1640                  * to release them before exiting: hence setting ncrfree
1641                  * to NULL
1642                  */
1643                 if (cred != cr)
1644                         ncrfree = NULL;
1645                 if ((acc & res.resok.access) != acc) {
```



```
1646                         /*
1647                          * If the cred can be adjusted, try again
1648                          * with the new cred.
1649                          */
1650                         if (ncr != NULL) {
1651                                 cred = ncr;
1652                                 ncr = NULL;
1653                                 goto tryagain;
1654                         }
1655                         error = EACCES;
1656                 }
1657         } else {
1658                 nfs3_cache_post_op_attr(vp, &res.resfail.obj_attributes, t, cr);
1659                 PURGE_STALE_FH(error, vp, cr);
1660         }

1662         if (ncrfree != NULL)
1663                 crfree(ncrfree);

1665         return (error);
1666 }

1668 static int nfs3_do_symlink_cache = 1;

1670 /* ARGSUSED */
1671 static int
1672 nfs3_readlink(vnode_t *vp, struct uio *uiop, cred_t *cr, caller_context_t *ct)
1673 {
1674         int error;
1675         READLINK3args args;
1676         READLINK3res res;
1677         nfspath3 resdata_backup;
1678         rnode_t *rp;
1679         int douprintf;
1680         int len;
1681         failinfo_t fi;
1682         hrtime_t t;

1684         /*
1685          * Can't readlink anything other than a symbolic link.
1686          */
1687         if (vp->v_type != VLNK)
1688                 return (EINVAL);
1689         if (nfs_zone() != VTOMI(vp)->mi_zone)
1690                 return (EIO);

1692         rp = VTOR(vp);
1693         if (nfs3_do_symlink_cache && rp->r_symlink.contents != NULL) {
1694                 error = nfs3_validate_caches(vp, cr);
1695                 if (error)
1696                         return (error);
1697                 mutex_enter(&rp->r_statelock);
1698                 if (rp->r_symlink.contents != NULL) {
1699                         error = uiomove(rp->r_symlink.contents,
1700                             rp->r_symlink.len, UIO_READ, uiop);
1701                         mutex_exit(&rp->r_statelock);
1702                         return (error);
1703                 }
1704                 mutex_exit(&rp->r_statelock);
1705         }

1707         args.symlink = *VTOFH3(vp);
1708         fi.vp = vp;
1709         fi.fhp = (caddr_t)&args.symlink;
1710         fi.copyproc = nfs3copyfh;
1711         fi.lookupproc = nfs3lookup;
```

```
1712         fi.xattrdirproc = acl_getxattrdir3;

1714         res.resok.data = kmem_alloc(MAXPATHLEN, KM_SLEEP);

1716         resdata_backup = res.resok.data;

1718         douprintf = 1;

1720         t = gethrtime();

1722         error = rfs3call(VTOMI(vp), NFSPROC3_READLINK,
1723             xdr_READLINK3args, (caddr_t)&args,
1724             xdr_READLINK3res, (caddr_t)&res, cr,
1725             &douprintf, &res.status, 0, &fi);

1727         if (res.resok.data == nfs3nametoolong)
1728                 error = EINVAL;

1730         if (error) {
1731                 kmem_free(resdata_backup, MAXPATHLEN);
1732                 return (error);
1733         }

1735         error = geterrno3(res.status);
1736         if (!error) {
1737                 nfs3_cache_post_op_attr(vp, &res.resok.symlink_attributes, t,
1738                     cr);
1739                 len = strlen(res.resok.data);
1740                 error = uiomove(res.resok.data, len, UIO_READ, uiop);
1741                 if (nfs3_do_symlink_cache && rp->r_symlink.contents == NULL) {
1742                         mutex_enter(&rp->r_statelock);
1743                                 if (rp->r_symlink.contents == NULL) {
1744                                 rp->r_symlink.contents = res.resok.data;
1745                                 rp->r_symlink.len = len;
1746                                 rp->r_symlink.size = MAXPATHLEN;
1747                                 mutex_exit(&rp->r_statelock);
1748                         } else {
1749                                 mutex_exit(&rp->r_statelock);

1751                                 kmem_free((void *)res.resok.data, MAXPATHLEN);
1752                         }
1753                 } else {
1754                         kmem_free((void *)res.resok.data, MAXPATHLEN);
1755                 }
1756         } else {
1757                 nfs3_cache_post_op_attr(vp,
1758                     &res.resfail.symlink_attributes, t, cr);
1759                 PURGE_STALE_FH(error, vp, cr);

1761                 kmem_free((void *)res.resok.data, MAXPATHLEN);

1763         }

1765         /*
1766          * The over the wire error for attempting to readlink something
1767          * other than a symbolic link is ENXIO.  However, we need to
1768          * return EINVAL instead of ENXIO, so we map it here.
1769          */
1770         return (error == ENXIO ? EINVAL : error);
1771 }

1773 /*
1774  * Flush local dirty pages to stable storage on the server.
1775  *
1776  * If FNODSYNC is specified, then there is nothing to do because
1777  * metadata changes are not cached on the client before being
```



```
1778  * sent to the server.
1779  */
1780 /* ARGSUSED */
1781 static int
1782 nfs3_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
1783 {
1784         int error;

1786         if ((syncflag & FNODSYNC) || IS_SWAPVP(vp))
1787                 return (0);
1788         if (nfs_zone() != VTOMI(vp)->mi_zone)
1789                 return (EIO);

1791         error = nfs3_putpage_commit(vp, (offset_t)0, 0, cr);
1792         if (!error)
1793                 error = VTOR(vp)->r_error;
1794         return (error);
1795 }

1797 /*
1798  * Weirdness: if the file was removed or the target of a rename
1799  * operation while it was open, it got renamed instead.  Here we
1800  * remove the renamed file.
1801  */
1802 /* ARGSUSED */
1803 static void
1804 nfs3_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1805 {
1806         rnode_t *rp;

1808         ASSERT(vp != DNLC_NO_VNODE);

1810         /*
1811          * If this is coming from the wrong zone, we let someone in the right
1812          * zone take care of it asynchronously.  We can get here due to
1813          * VN_RELE() being called from pageout() or fsflush().  This call may
1814          * potentially turn into an expensive no-op if, for instance, v_count
1815          * gets incremented in the meantime, but it's still correct.
1816          */
1817         if (nfs_zone() != VTOMI(vp)->mi_zone) {
1818                 nfs_async_inactive(vp, cr, nfs3_inactive);
1819                 return;
1820         }

1822         rp = VTOR(vp);
1823 redo:
1824         if (rp->r_unldvp != NULL) {
1825                 /*
1826                  * Save the vnode pointer for the directory where the
1827                  * unlinked-open file got renamed, then set it to NULL
1828                  * to prevent another thread from getting here before
1829                  * we're done with the remove.  While we have the
1830                  * statelock, make local copies of the pertinent rnode
1831                  * fields.  If we weren't to do this in an atomic way, the
1832                  * the unl* fields could become inconsistent with respect
1833                  * to each other due to a race condition between this
1834                  * code and nfs_remove().  See bug report 1034328.
1835                  */
1836                 mutex_enter(&rp->r_statelock);
1837                 if (rp->r_unldvp != NULL) {
1838                         vnode_t *unldvp;
1839                         char *unlname;
1840                         cred_t *unlcred;
1841                         REMOVE3args args;
1842                         REMOVE3res res;
1843                         int douprintf;
```

```
1844                    int error;
1845                    hrtime_t t;

1847                    unldvp = rp->r_unldvp;
1848                    rp->r_unldvp = NULL;
1849                    unlname = rp->r_unlname;
1850                    rp->r_unlname = NULL;
1851                    unlcred = rp->r_unlcred;
1852                    rp->r_unlcred = NULL;
1853                    mutex_exit(&rp->r_statelock);

1855                    /*
1856                     * If there are any dirty pages left, then flush
1857                     * them.  This is unfortunate because they just
1858                     * may get thrown away during the remove operation,
1859                     * but we have to do this for correctness.
1860                     */
1861                    if (vn_has_cached_data(vp) &&
1862                        ((rp->r_flags & RDIRTY) || rp->r_count > 0)) {
1863                            ASSERT(vp->v_type != VCHR);
1864                            error = nfs3_putpage(vp, (offset_t)0, 0, 0,
1865                                cr, ct);
1866                            if (error) {
1867                                    mutex_enter(&rp->r_statelock);
1868                                    if (!rp->r_error)
1869                                            rp->r_error = error;
1870                                    mutex_exit(&rp->r_statelock);
1871                            }
1872                    }

1874                    /*
1875                     * Do the remove operation on the renamed file
1876                     */
1877                    setdiropargs3(&args.object, unlname, unldvp);

1879                    douprintf = 1;

1881                    t = gethrtime();

1883                    error = rfs3call(VTOMI(unldvp), NFSPROC3_REMOVE,
1884                        xdr_diropargs3, (caddr_t)&args,
1885                        xdr_REMOVE3res, (caddr_t)&res, unlcred,
1886                        &douprintf, &res.status, 0, NULL);

1888                    if (error) {
1889                            PURGE_ATTRCACHE(unldvp);
1890                    } else {
1891                            error = geterrno3(res.status);
1892                            if (!error) {
1893                                    nfs3_cache_wcc_data(unldvp,
1894                                        &res.resok.dir_wcc, t, cr);
1895                                    if (HAVE_RDDIR_CACHE(VTOR(unldvp)))
1896                                            nfs_purge_rddir_cache(unldvp);
1897                            } else {
1898                                    nfs3_cache_wcc_data(unldvp,
1899                                        &res.resfail.dir_wcc, t, cr);
1900                                    PURGE_STALE_FH(error, unldvp, cr);
1901                            }
1902                    }

1904                    /*
1905                     * Release stuff held for the remove
1906                     */
1907                    VN_RELE(unldvp);
1908                    kmem_free(unlname, MAXNAMELEN);
1909                    crfree(unlcred);
```



```
1910                            goto redo;
1911                    }
1912                    mutex_exit(&rp->r_statelock);
1913            }

1915            rp_addfree(rp, cr);
1916    }

1918    /*
1919     * Remote file system operations having to do with directory manipulation.
1920     */

1922    /* ARGSUSED */
1923    static int
1924    nfs3_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
1925            int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
1926            int *direntflags, pathname_t *realpnp)
1927    {
1928            int error;
1929            vnode_t *vp;
1930            vnode_t *avp = NULL;
1931            rnode_t *drp;

1933            if (nfs_zone() != VTOMI(dvp)->mi_zone)
1934                    return (EPERM);

1936            drp = VTOR(dvp);

1938            /*
1939             * Are we looking up extended attributes?  If so, "dvp" is
1940             * the file or directory for which we want attributes, and
1941             * we need a lookup of the hidden attribute directory
1942             * before we lookup the rest of the path.
1943             */
1944            if (flags & LOOKUP_XATTR) {
1945                    bool_t cflag = ((flags & CREATE_XATTR_DIR) != 0);
1946                    mntinfo_t *mi;

1948                    mi = VTOMI(dvp);
1949                    if (!(mi->mi_flags & MI_EXTATTR))
1950                            return (EINVAL);

1952                    if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR(dvp)))
1953                            return (EINTR);

1955                    (void) nfs3lookup_dnlc(dvp, XATTR_DIR_NAME, &avp, cr);
1956                    if (avp == NULL)
1957                            error = acl_getxattrdir3(dvp, &avp, cflag, cr, 0);
1958                    else
1959                            error = 0;

1961                    nfs_rw_exit(&drp->r_rwlock);

1963                    if (error) {
1964                            if (mi->mi_flags & MI_EXTATTR)
1965                                    return (error);
1966                            return (EINVAL);
1967                    }
1968                    dvp = avp;
1969                    drp = VTOR(dvp);
1970            }

1972            if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR(dvp))) {
1973                    error = EINTR;
1974                    goto out;
1975            }
```

```
1977         error = nfs3lookup(dvp, nm, vpp, pnp, flags, rdir, cr, 0);

1979         nfs_rw_exit(&drp->r_rwlock);

1981         /*
1982          * If vnode is a device, create special vnode.
1983          */
1984         if (!error && IS_DEVVP(*vpp)) {
1985                 vp = *vpp;
1986                 *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
1987                 VN_RELE(vp);
1988         }

1990 out:
1991         if (avp != NULL)
1992                 VN_RELE(avp);

1994         return (error);
1995 }

1997 static int nfs3_lookup_neg_cache = 1;

1999 #ifdef DEBUG
2000 static int nfs3_lookup_dnlc_hits = 0;
2001 static int nfs3_lookup_dnlc_misses = 0;
2002 static int nfs3_lookup_dnlc_neg_hits = 0;
2003 static int nfs3_lookup_dnlc_disappears = 0;
2004 static int nfs3_lookup_dnlc_lookups = 0;
2005 #endif

2007 /* ARGSUSED */
2008 int
2009 nfs3lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
2010         int flags, vnode_t *rdir, cred_t *cr, int rfscall_flags)
2011 {
2012         int error;
2013         rnode_t *drp;

2015         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);
2016         /*
2017          * If lookup is for "", just return dvp.  Don't need
2018          * to send it over the wire, look it up in the dnlc,
2019          * or perform any access checks.
2020          */
2021         if (*nm == '\0') {
2022                 VN_HOLD(dvp);
2023                 *vpp = dvp;
2024                 return (0);
2025         }

2027         /*
2028          * Can't do lookups in non-directories.
2029          */
2030         if (dvp->v_type != VDIR)
2031                 return (ENOTDIR);

2033         /*
2034          * If we're called with RFSCALL_SOFT, it's important that
2035          * the only rfscall is one we make directly; if we permit
2036          * an access call because we're looking up "." or validating
2037          * a dnlc hit, we'll deadlock because that rfscall will not
2038          * have the RFSCALL_SOFT set.
2039          */
2040         if (rfscall_flags & RFSCALL_SOFT)
2041                 goto callit;
```



```
2043         /*
2044          * If lookup is for ".", just return dvp.  Don't need
2045          * to send it over the wire or look it up in the dnlc,
2046          * just need to check access.
2047          */
2048         if (strcmp(nm, ".") == 0) {
2049                 error = nfs3_access(dvp, VEXEC, 0, cr, NULL);
2050                 if (error)
2051                         return (error);
2052                 VN_HOLD(dvp);
2053                 *vpp = dvp;
2054                 return (0);
2055         }

2057         drp = VTOR(dvp);
2058         if (!(drp->r_flags & RLOOKUP)) {
2059                 mutex_enter(&drp->r_statelock);
2060                 drp->r_flags |= RLOOKUP;
2061                 mutex_exit(&drp->r_statelock);
2062         }

2064         /*
2065          * Lookup this name in the DNLC.  If there was a valid entry,
2066          * then return the results of the lookup.
2067          */
2068         error = nfs3lookup_dnlc(dvp, nm, vpp, cr);
2069         if (error || *vpp != NULL)
2070                 return (error);

2072 callit:
2073         error = nfs3lookup_otw(dvp, nm, vpp, cr, rfscall_flags);

2075         return (error);
2076 }

2078 static int
2079 nfs3lookup_dnlc(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr)
2080 {
2081         int error;
2082         vnode_t *vp;

2084         ASSERT(*nm != '\0');
2085         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);
2086         /*
2087          * Lookup this name in the DNLC.  If successful, then validate
2088          * the caches and then recheck the DNLC.  The DNLC is rechecked
2089          * just in case this entry got invalidated during the call
2090          * to nfs3_validate_caches.
2091          *
2092          * An assumption is being made that it is safe to say that a
2093          * file exists which may not on the server.  Any operations to
2094          * the server will fail with ESTALE.
2095          */
2096 #ifdef DEBUG
2097         nfs3_lookup_dnlc_lookups++;
2098 #endif
2099         vp = dnlc_lookup(dvp, nm);
2100         if (vp != NULL) {
2101                 VN_RELE(vp);
2102                 if (vp == DNLC_NO_VNODE && !vn_is_readonly(dvp)) {
2103                         PURGE_ATTRCACHE(dvp);
2104                 }
2105                 error = nfs3_validate_caches(dvp, cr);
2106                 if (error)
2107                         return (error);
```

```
2108                 vp = dnlc_lookup(dvp, nm);
2109                 if (vp != NULL) {
2110                         error = nfs3_access(dvp, VEXEC, 0, cr, NULL);
2111                         if (error) {
2112                                 VN_RELE(vp);
2113                                 return (error);
2114                         }
2115                         if (vp == DNLC_NO_VNODE) {
2116                                 VN_RELE(vp);
2117 #ifdef DEBUG
2118                                 nfs3_lookup_dnlc_neg_hits++;
2119 #endif
2120                                 return (ENOENT);
2121                         }
2122                         *vpp = vp;
2123 #ifdef DEBUG
2124                         nfs3_lookup_dnlc_hits++;
2125 #endif
2126                         return (0);
2127                 }
2128 #ifdef DEBUG
2129                 nfs3_lookup_dnlc_disappears++;
2130 #endif
2131         }
2132 #ifdef DEBUG
2133         else
2134                 nfs3_lookup_dnlc_misses++;
2135 #endif

2137         *vpp = NULL;

2139         return (0);
2140 }

2142 static int
2143 nfs3lookup_otw(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
2144         int rfscall_flags)
2145 {
2146         int error;
2147         LOOKUP3args args;
2148         LOOKUP3vres res;
2149         int douprintf;
2150         struct vattr vattr;
2151         struct vattr dvattr;
2152         vnode_t *vp;
2153         failinfo_t fi;
2154         hrtime_t t;

2156         ASSERT(*nm != '\0');
2157         ASSERT(dvp->v_type == VDIR);
2158         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);

2160         setdiropargs3(&args.what, nm, dvp);

2162         fi.vp = dvp;
2163         fi.fhp = (caddr_t)&args.what.dir;
2164         fi.copyproc = nfs3copyfh;
2165         fi.lookupproc = nfs3lookup;
2166         fi.xattrdirproc = acl_getxattrdir3;
2167         res.obj_attributes.fres.vp = dvp;
2168         res.obj_attributes.fres.vap = &vattr;
2169         res.dir_attributes.fres.vp = dvp;
2170         res.dir_attributes.fres.vap = &dvattr;

2172         douprintf = 1;
```



```
2174         t = gethrtime();

2176         error = rfs3call(VTOMI(dvp), NFSPROC3_LOOKUP,
2177             xdr_diropargs3, (caddr_t)&args,
2178             xdr_LOOKUP3vres, (caddr_t)&res, cr,
2179             &douprintf, &res.status, rfscall_flags, &fi);

2181         if (error)
2182                 return (error);

2184         nfs3_cache_post_op_vattr(dvp, &res.dir_attributes, t, cr);

2186         error = geterrno3(res.status);
2187         if (error) {
2188                 PURGE_STALE_FH(error, dvp, cr);
2189                 if (error == ENOENT && nfs3_lookup_neg_cache)
2190                         dnlc_enter(dvp, nm, DNLC_NO_VNODE);
2191                 return (error);
2192         }

2194         if (res.obj_attributes.attributes) {
2195                 vp = makenfs3node_va(&res.object, res.obj_attributes.fres.vap,
2196                     dvp->v_vfsp, t, cr, VTOR(dvp)->r_path, nm);
2197         } else {
2198                 vp = makenfs3node_va(&res.object, NULL,
2199                     dvp->v_vfsp, t, cr, VTOR(dvp)->r_path, nm);
2200                 if (vp->v_type == VNON) {
2201                         vattr.va_mask = AT_TYPE;
2202                         error = nfs3getattr(vp, &vattr, cr);
2203                         if (error) {
2204                                 VN_RELE(vp);
2205                                 return (error);
2206                         }
2207                         vp->v_type = vattr.va_type;
2208                 }
2209         }

2211         if (!(rfscall_flags & RFSCALL_SOFT))
2212                 dnlc_update(dvp, nm, vp);

2214         *vpp = vp;

2216         return (error);
2217 }

2219 #ifdef DEBUG
2220 static int nfs3_create_misses = 0;
2221 #endif

2223 /* ARGSUSED */
2224 static int
2225 nfs3_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
2226         int mode, vnode_t **vpp, cred_t *cr, int lfaware, caller_context_t *ct,
2227         vsecattr_t *vsecp)
2228 {
2229         int error;
2230         vnode_t *vp;
2231         rnode_t *rp;
2232         struct vattr vattr;
2233         rnode_t *drp;
2234         vnode_t *tempvp;

2236         drp = VTOR(dvp);
2237         if (nfs_zone() != VTOMI(dvp)->mi_zone)
2238                 return (EPERM);
2239         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
```

```
2240                 return (EINTR);

2242 top:
2243         /*
2244          * We make a copy of the attributes because the caller does not
2245          * expect us to change what va points to.
2246          */
2247         vattr = *va;

2249         /*
2250          * If the pathname is "", just use dvp.  Don't need
2251          * to send it over the wire, look it up in the dnlc,
2252          * or perform any access checks.
2253          */
2254         if (*nm == '\0') {
2255                 error = 0;
2256                 VN_HOLD(dvp);
2257                 vp = dvp;
2258         /*
2259          * If the pathname is ".", just use dvp.  Don't need
2260          * to send it over the wire or look it up in the dnlc,
2261          * just need to check access.
2262          */
2263         } else if (strcmp(nm, ".") == 0) {
2264                 error = nfs3_access(dvp, VEXEC, 0, cr, ct);
2265                 if (error) {
2266                         nfs_rw_exit(&drp->r_rwlock);
2267                         return (error);
2268                 }
2269                 VN_HOLD(dvp);
2270                 vp = dvp;
2271         /*
2272          * We need to go over the wire, just to be sure whether the
2273          * file exists or not.  Using the DNLC can be dangerous in
2274          * this case when making a decision regarding existence.
2275          */
2276         } else {
2277                 error = nfs3lookup_otw(dvp, nm, &vp, cr, 0);
2278         }
2279         if (!error) {
2280                 if (exclusive == EXCL)
2281                         error = EEXIST;
2282                 else if (vp->v_type == VDIR && (mode & VWRITE))
2283                         error = EISDIR;
2284                 else {
2285                         /*
2286                          * If vnode is a device, create special vnode.
2287                          */
2288                         if (IS_DEVVP(vp)) {
2289                                 tempvp = vp;
2290                                 vp = specvp(vp, vp->v_rdev, vp->v_type, cr);
2291                                 VN_RELE(tempvp);
2292                         }
2293                         if (!(error = VOP_ACCESS(vp, mode, 0, cr, ct))) {
2294                                 if ((vattr.va_mask & AT_SIZE) &&
2295                                     vp->v_type == VREG) {
2296                                         rp = VTOR(vp);
2297                                         /*
2298                                          * Check here for large file handled
2299                                          * by LF-unaware process (as
2300                                          * ufs_create() does)
2301                                          */
2302                                         if (!(lfaware & FOFFMAX)) {
2303                                                 mutex_enter(&rp->r_statelock);
2304                                                 if (rp->r_size > MAXOFF32_T)
2305                                                         error = EOVERFLOW;
```



```
2306                                                 mutex_exit(&rp->r_statelock);
2307                                         }
2308                                         if (!error) {
2309                                                 vattr.va_mask = AT_SIZE;
2310                                                 error = nfs3setattr(vp,
2311                                                     &vattr, 0, cr);

2313                                                 /*
2314                                                  * Existing file was truncated;
2315                                                  * emit a create event.
2316                                                  */
2317                                                 vnevent_create(vp, ct);
2318                                         }
2319                                 }
2320                         }
2321                 }
2322                 nfs_rw_exit(&drp->r_rwlock);
2323                 if (error) {
2324                         VN_RELE(vp);
2325                 } else {
2326                         *vpp = vp;
2327                 }

2329                 return (error);
2330         }

2332         dnlc_remove(dvp, nm);

2334         /*
2335          * Decide what the group-id of the created file should be.
2336          * Set it in attribute list as advisory...
2337          */
2338         error = setdirgid(dvp, &vattr.va_gid, cr);
2339         if (error) {
2340                 nfs_rw_exit(&drp->r_rwlock);
2341                 return (error);
2342         }
2343         vattr.va_mask |= AT_GID;

2345         ASSERT(vattr.va_mask & AT_TYPE);
2346         if (vattr.va_type == VREG) {
2347                 ASSERT(vattr.va_mask & AT_MODE);
2348                 if (MANDMODE(vattr.va_mode)) {
2349                         nfs_rw_exit(&drp->r_rwlock);
2350                         return (EACCES);
2351                 }
2352                 error = nfs3create(dvp, nm, &vattr, exclusive, mode, vpp, cr,
2353                     lfaware);
2354                 /*
2355                  * If this is not an exclusive create, then the CREATE
2356                  * request will be made with the GUARDED mode set.  This
2357                  * means that the server will return EEXIST if the file
2358                  * exists.  The file could exist because of a retransmitted
2359                  * request.  In this case, we recover by starting over and
2360                  * checking to see whether the file exists.  This second
2361                  * time through it should and a CREATE request will not be
2362                  * sent.
2363                  *
2364                  * This handles the problem of a dangling CREATE request
2365                  * which contains attributes which indicate that the file
2366                  * should be truncated.  This retransmitted request could
2367                  * possibly truncate valid data in the file if not caught
2368                  * by the duplicate request mechanism on the server or if
2369                  * not caught by other means.  The scenario is:
2370                  *
2371                  * Client transmits CREATE request with size = 0
```

```
2372                  * Client times out, retransmits request.
2373                  * Response to the first request arrives from the server
2374                  *  and the client proceeds on.
2375                  * Client writes data to the file.
2376                  * The server now processes retransmitted CREATE request
2377                  *  and truncates file.
2378                  *
2379                  * The use of the GUARDED CREATE request prevents this from
2380                  * happening because the retransmitted CREATE would fail
2381                  * with EEXIST and would not truncate the file.
2382                  */
2383                 if (error == EEXIST && exclusive == NONEXCL) {
2384 #ifdef DEBUG
2385                         nfs3_create_misses++;
2386 #endif
2387                         goto top;
2388                 }
2389                 nfs_rw_exit(&drp->r_rwlock);
2390                 return (error);
2391         }
2392         error = nfs3mknod(dvp, nm, &vattr, exclusive, mode, vpp, cr);
2393         nfs_rw_exit(&drp->r_rwlock);
2394         return (error);
2395 }

2397 /* ARGSUSED */
2398 static int
2399 nfs3create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
2400         int mode, vnode_t **vpp, cred_t *cr, int lfaware)
2401 {
2402         int error;
2403         CREATE3args args;
2404         CREATE3res res;
2405         int douprintf;
2406         vnode_t *vp;
2407         struct vattr vattr;
2408         nfstime3 *verfp;
2409         rnode_t *rp;
2410         timestruc_t now;
2411         hrtime_t t;

2413         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);
2414         setdiropargs3(&args.where, nm, dvp);
2415         if (exclusive == EXCL) {
2416                 args.how.mode = EXCLUSIVE;
2417                 /*
2418                  * Construct the create verifier.  This verifier needs
2419                  * to be unique between different clients.  It also needs
2420                  * to vary for each exclusive create request generated
2421                  * from the client to the server.
2422                  *
2423                  * The first attempt is made to use the hostid and a
2424                  * unique number on the client.  If the hostid has not
2425                  * been set, the high resolution time that the exclusive
2426                  * create request is being made is used.  This will work
2427                  * unless two different clients, both with the hostid
2428                  * not set, attempt an exclusive create request on the
2429                  * same file, at exactly the same clock time.  The
2430                  * chances of this happening seem small enough to be
2431                  * reasonable.
2432                  */
2433                 verfp = (nfstime3 *)&args.how.createhow3_u.verf;
2434                 verfp->seconds = zone_get_hostid(NULL);
2435                 if (verfp->seconds != 0)
2436                         verfp->nseconds = newnum();
2437                 else {
```



```
2438                         gethrestime(&now);
2439                         verfp->seconds = now.tv_sec;
2440                         verfp->nseconds = now.tv_nsec;
2441                 }
2442                 /*
2443                  * Since the server will use this value for the mtime,
2444                  * make sure that it can't overflow. Zero out the MSB.
2445                  * The actual value does not matter here, only its uniqeness.
2446                  */
2447                 verfp->seconds %= INT32_MAX;
2448         } else {
2449                 /*
2450                  * Issue the non-exclusive create in guarded mode.  This
2451                  * may result in some false EEXIST responses for
2452                  * retransmitted requests, but these will be handled at
2453                  * a higher level.  By using GUARDED, duplicate requests
2454                  * to do file truncation and possible access problems
2455                  * can be avoided.
2456                  */
2457                 args.how.mode = GUARDED;
2458                 error = vattr_to_sattr3(va,
2459                     &args.how.createhow3_u.obj_attributes);
2460                 if (error) {
2461                         /* req time field(s) overflow - return immediately */
2462                         return (error);
2463                 }
2464         }

2466         douprintf = 1;

2468         t = gethrtime();

2470         error = rfs3call(VTOMI(dvp), NFSPROC3_CREATE,
2471             xdr_CREATE3args, (caddr_t)&args,
2472             xdr_CREATE3res, (caddr_t)&res, cr,
2473             &douprintf, &res.status, 0, NULL);

2475         if (error) {
2476                 PURGE_ATTRCACHE(dvp);
2477                 return (error);
2478         }

2480         error = geterrno3(res.status);
2481         if (!error) {
2482                 nfs3_cache_wcc_data(dvp, &res.resok.dir_wcc, t, cr);
2483                 if (HAVE_RDDIR_CACHE(VTOR(dvp)))
2484                         nfs_purge_rddir_cache(dvp);

2486                 /*
2487                  * On exclusive create the times need to be explicitly
2488                  * set to clear any potential verifier that may be stored
2489                  * in one of these fields (see comment below).  This
2490                  * is done here to cover the case where no post op attrs
2491                  * were returned or a 'invalid' time was returned in
2492                  * the attributes.
2493                  */
2494                 if (exclusive == EXCL)
2495                         va->va_mask |= (AT_MTIME | AT_ATIME);

2497                 if (!res.resok.obj.handle_follows) {
2498                         error = nfs3lookup(dvp, nm, &vp, NULL, 0, NULL, cr, 0);
2499                         if (error)
2500                                 return (error);
2501                 } else {
2502                         if (res.resok.obj_attributes.attributes) {
2503                                 vp = makenfs3node(&res.resok.obj.handle,
```

```
                                    &res.resok.obj_attributes.attr,
                                    dvp->v_vfsp, t, cr, NULL, NULL);
                        } else {
                                vp = makenfs3node(&res.resok.obj.handle, NULL,
                                    dvp->v_vfsp, t, cr, NULL, NULL);

                                /*
                                 * On an exclusive create, it is possible
                                 * that attributes were returned but those
                                 * postop attributes failed to decode
                                 * properly.  If this is the case,
                                 * then most likely the atime or mtime
                                 * were invalid for our client; this
                                 * is caused by the server storing the
                                 * create verifier in one of the time
                                 * fields(most likely mtime).
                                 * So... we are going to setattr just the
                                 * atime/mtime to clear things up.
                                 */
                                if (exclusive == EXCL) {
                                        if (error =
                                            nfs3excl_create_settimes(vp,
                                            va, cr)) {
                                                /*
                                                 * Setting the times failed.
                                                 * Remove the file and return
                                                 * the error.
                                                 */
                                                VN_RELE(vp);
                                                (void) nfs3_remove(dvp,
                                                    nm, cr, NULL, 0);
                                                return (error);
                                        }
                                }

                                /*
                                 * This handles the non-exclusive case
                                 * and the exclusive case where no post op
                                 * attrs were returned.
                                 */
                                if (vp->v_type == VNON) {
                                        vattr.va_mask = AT_TYPE;
                                        error = nfs3getattr(vp, &vattr, cr);
                                        if (error) {
                                                VN_RELE(vp);
                                                return (error);
                                        }
                                        vp->v_type = vattr.va_type;
                                }
                        }
                        dnlc_update(dvp, nm, vp);
                }

                rp = VTOR(vp);

                /*
                 * Check here for large file handled by
                 * LF-unaware process (as ufs_create() does)
                 */
                if ((va->va_mask & AT_SIZE) && vp->v_type == VREG &&
                    !(lfaware & FOFFMAX)) {
                        mutex_enter(&rp->r_statelock);
                        if (rp->r_size > MAXOFF32_T) {
                                mutex_exit(&rp->r_statelock);
                                VN_RELE(vp);
                                return (EOVERFLOW);
```



```
                        }
                        mutex_exit(&rp->r_statelock);
                }

                if (exclusive == EXCL &&
                    (va->va_mask & ~(AT_GID | AT_SIZE))) {
                        /*
                         * If doing an exclusive create, then generate
                         * a SETATTR to set the initial attributes.
                         * Try to set the mtime and the atime to the
                         * server's current time.  It is somewhat
                         * expected that these fields will be used to
                         * store the exclusive create cookie.  If not,
                         * server implementors will need to know that
                         * a SETATTR will follow an exclusive create
                         * and the cookie should be destroyed if
                         * appropriate. This work may have been done
                         * earlier in this function if post op attrs
                         * were not available.
                         *
                         * The AT_GID and AT_SIZE bits are turned off
                         * so that the SETATTR request will not attempt
                         * to process these.  The gid will be set
                         * separately if appropriate.  The size is turned
                         * off because it is assumed that a new file will
                         * be created empty and if the file wasn't empty,
                         * then the exclusive create will have failed
                         * because the file must have existed already.
                         * Therefore, no truncate operation is needed.
                         */
                        va->va_mask &= ~(AT_GID | AT_SIZE);
                        error = nfs3setattr(vp, va, 0, cr);
                        if (error) {
                                /*
                                 * Couldn't correct the attributes of
                                 * the newly created file and the
                                 * attributes are wrong.  Remove the
                                 * file and return an error to the
                                 * application.
                                 */
                                VN_RELE(vp);
                                (void) nfs3_remove(dvp, nm, cr, NULL, 0);
                                return (error);
                        }
                }

                if (va->va_gid != rp->r_attr.va_gid) {
                        /*
                         * If the gid on the file isn't right, then
                         * generate a SETATTR to attempt to change
                         * it.  This may or may not work, depending
                         * upon the server's semantics for allowing
                         * file ownership changes.
                         */
                        va->va_mask = AT_GID;
                        (void) nfs3setattr(vp, va, 0, cr);
                }

                /*
                 * If vnode is a device create special vnode
                 */
                if (IS_DEVVP(vp)) {
                        *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
                        VN_RELE(vp);
                } else
                        *vpp = vp;
```

```
2636         } else {
2637                 nfs3_cache_wcc_data(dvp, &res.resfail.dir_wcc, t, cr);
2638                 PURGE_STALE_FH(error, dvp, cr);
2639         }

2641         return (error);
2642 }

2644 /*
2645  * Special setattr function to take care of rest of atime/mtime
2646  * after successful exclusive create.  This function exists to avoid
2647  * handling attributes from the server; exclusive the atime/mtime fields
2648  * may be 'invalid' in client's view and therefore can not be trusted.
2649  */
2650 static int
2651 nfs3excl_create_settimes(vnode_t *vp, struct vattr *vap, cred_t *cr)
2652 {
2653         int error;
2654         uint_t mask;
2655         SETATTR3args args;
2656         SETATTR3res res;
2657         int douprintf;
2658         rnode_t *rp;
2659         hrtime_t t;

2661         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);
2662         /* save the caller's mask so that it can be reset later */
2663         mask = vap->va_mask;

2665         rp = VTOR(vp);

2667         args.object = *RTOFH3(rp);
2668         args.guard.check = FALSE;

2670         /* Use the mask to initialize the arguments */
2671         vap->va_mask = 0;
2672         error = vattr_to_sattr3(vap, &args.new_attributes);

2674         /* We want to set just atime/mtime on this request */
2675         args.new_attributes.atime.set_it = SET_TO_SERVER_TIME;
2676         args.new_attributes.mtime.set_it = SET_TO_SERVER_TIME;

2678         douprintf = 1;

2680         t = gethrtime();

2682         error = rfs3call(VTOMI(vp), NFSPROC3_SETATTR,
2683             xdr_SETATTR3args, (caddr_t)&args,
2684             xdr_SETATTR3res, (caddr_t)&res, cr,
2685             &douprintf, &res.status, 0, NULL);

2687         if (error) {
2688                 vap->va_mask = mask;
2689                 return (error);
2690         }

2692         error = geterrno3(res.status);
2693         if (!error) {
2694                 /*
2695                  * It is important to pick up the attributes.
2696                  * Since this is the exclusive create path, the
2697                  * attributes on the initial create were ignored
2698                  * and we need these to have the correct info.
2699                  */
2700                 nfs3_cache_wcc_data(vp, &res.resok.obj_wcc, t, cr);
2701                 /*
```



```
2702                  * No need to do the atime/mtime work again so clear
2703                  * the bits.
2704                  */
2705                 mask &= ~(AT_ATIME | AT_MTIME);
2706         } else {
2707                 nfs3_cache_wcc_data(vp, &res.resfail.obj_wcc, t, cr);
2708         }

2710         vap->va_mask = mask;

2712         return (error);
2713 }

2715 /* ARGSUSED */
2716 static int
2717 nfs3mknod(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
2718         int mode, vnode_t **vpp, cred_t *cr)
2719 {
2720         int error;
2721         MKNOD3args args;
2722         MKNOD3res res;
2723         int douprintf;
2724         vnode_t *vp;
2725         struct vattr vattr;
2726         hrtime_t t;

2728         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);
2729         switch (va->va_type) {
2730         case VCHR:
2731         case VBLK:
2732                 setdiropargs3(&args.where, nm, dvp);
2733                 args.what.type = (va->va_type == VCHR) ? NF3CHR : NF3BLK;
2734                 error = vattr_to_sattr3(va,
2735                     &args.what.mknoddata3_u.device.dev_attributes);
2736                 if (error) {
2737                         /* req time field(s) overflow - return immediately */
2738                         return (error);
2739                 }
2740                 args.what.mknoddata3_u.device.spec.specdata1 =
2741                     getmajor(va->va_rdev);
2742                 args.what.mknoddata3_u.device.spec.specdata2 =
2743                     getminor(va->va_rdev);
2744                 break;

2746         case VFIFO:
2747         case VSOCK:
2748                 setdiropargs3(&args.where, nm, dvp);
2749                 args.what.type = (va->va_type == VFIFO) ? NF3FIFO : NF3SOCK;
2750                 error = vattr_to_sattr3(va,
2751                     &args.what.mknoddata3_u.pipe_attributes);
2752                 if (error) {
2753                         /* req time field(s) overflow - return immediately */
2754                         return (error);
2755                 }
2756                 break;

2758         default:
2759                 return (EINVAL);
2760         }

2762         douprintf = 1;

2764         t = gethrtime();

2766         error = rfs3call(VTOMI(dvp), NFSPROC3_MKNOD,
2767             xdr_MKNOD3args, (caddr_t)&args,
```

```
2768             xdr_MKNOD3res, (caddr_t)&res, cr,
2769             &douprintf, &res.status, 0, NULL);

2771         if (error) {
2772                 PURGE_ATTRCACHE(dvp);
2773                 return (error);
2774         }

2776         error = geterrno3(res.status);
2777         if (!error) {
2778                 nfs3_cache_wcc_data(dvp, &res.resok.dir_wcc, t, cr);
2779                 if (HAVE_RDDIR_CACHE(VTOR(dvp)))
2780                         nfs_purge_rddir_cache(dvp);

2782                 if (!res.resok.obj.handle_follows) {
2783                         error = nfs3lookup(dvp, nm, &vp, NULL, 0, NULL, cr, 0);
2784                         if (error)
2785                                 return (error);
2786                 } else {
2787                         if (res.resok.obj_attributes.attributes) {
2788                                 vp = makenfs3node(&res.resok.obj.handle,
2789                                     &res.resok.obj_attributes.attr,
2790                                     dvp->v_vfsp, t, cr, NULL, NULL);
2791                         } else {
2792                                 vp = makenfs3node(&res.resok.obj.handle, NULL,
2793                                     dvp->v_vfsp, t, cr, NULL, NULL);
2794                                 if (vp->v_type == VNON) {
2795                                         vattr.va_mask = AT_TYPE;
2796                                         error = nfs3getattr(vp, &vattr, cr);
2797                                         if (error) {
2798                                                 VN_RELE(vp);
2799                                                 return (error);
2800                                         }
2801                                         vp->v_type = vattr.va_type;
2802                                 }

2804                         }
2805                         dnlc_update(dvp, nm, vp);
2806                 }

2808                 if (va->va_gid != VTOR(vp)->r_attr.va_gid) {
2809                         va->va_mask = AT_GID;
2810                         (void) nfs3setattr(vp, va, 0, cr);
2811                 }

2813                 /*
2814                  * If vnode is a device create special vnode
2815                  */
2816                 if (IS_DEVVP(vp)) {
2817                         *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
2818                         VN_RELE(vp);
2819                 } else
2820                         *vpp = vp;
2821         } else {
2822                 nfs3_cache_wcc_data(dvp, &res.resfail.dir_wcc, t, cr);
2823                 PURGE_STALE_FH(error, dvp, cr);
2824         }
2825         return (error);
2826 }

2828 /*
2829  * Weirdness: if the vnode to be removed is open
2830  * we rename it instead of removing it and nfs_inactive
2831  * will remove the new name.
2832  */
2833 /* ARGSUSED */
```



```
2834 static int
2835 nfs3_remove(vnode_t *dvp, char *nm, cred_t *cr, caller_context_t *ct, int flags)
2836 {
2837         int error;
2838         REMOVE3args args;
2839         REMOVE3res res;
2840         vnode_t *vp;
2841         char *tmpname;
2842         int douprintf;
2843         rnode_t *rp;
2844         rnode_t *drp;
2845         hrtime_t t;

2847         if (nfs_zone() != VTOMI(dvp)->mi_zone)
2848                 return (EPERM);
2849         drp = VTOR(dvp);
2850         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
2851                 return (EINTR);

2853         error = nfs3lookup(dvp, nm, &vp, NULL, 0, NULL, cr, 0);
2854         if (error) {
2855                 nfs_rw_exit(&drp->r_rwlock);
2856                 return (error);
2857         }

2859         if (vp->v_type == VDIR && secpolicy_fs_linkdir(cr, dvp->v_vfsp)) {
2860                 VN_RELE(vp);
2861                 nfs_rw_exit(&drp->r_rwlock);
2862                 return (EPERM);
2863         }

2865         /*
2866          * First just remove the entry from the name cache, as it
2867          * is most likely the only entry for this vp.
2868          */
2869         dnlc_remove(dvp, nm);

2871         /*
2872          * If the file has a v_count > 1 then there may be more than one
2873          * entry in the name cache due multiple links or an open file,
2874          * but we don't have the real reference count so flush all
2875          * possible entries.
2876          */
2877         if (vp->v_count > 1)
2878                 dnlc_purge_vp(vp);

2880         /*
2881          * Now we have the real reference count on the vnode
2882          */
2883         rp = VTOR(vp);
2884         mutex_enter(&rp->r_statelock);
2885         if (vp->v_count > 1 &&
2886             (rp->r_unldvp == NULL || strcmp(nm, rp->r_unlname) == 0)) {
2887                 mutex_exit(&rp->r_statelock);
2888                 tmpname = newname();
2889                 error = nfs3rename(dvp, nm, dvp, tmpname, cr, ct);
2890                 if (error)
2891                         kmem_free(tmpname, MAXNAMELEN);
2892                 else {
2893                         mutex_enter(&rp->r_statelock);
2894                         if (rp->r_unldvp == NULL) {
2895                                 VN_HOLD(dvp);
2896                                 rp->r_unldvp = dvp;
2897                                 if (rp->r_unlcred != NULL)
2898                                         crfree(rp->r_unlcred);
2899                                 crhold(cr);
```

```
2900                                rp->r_unlcred = cr;
2901                                rp->r_unlname = tmpname;
2902                        } else {
2903                                kmem_free(rp->r_unlname, MAXNAMELEN);
2904                                rp->r_unlname = tmpname;
2905                        }
2906                        mutex_exit(&rp->r_statelock);
2907                }
2908        } else {
2909                mutex_exit(&rp->r_statelock);
2910                /*
2911                 * We need to flush any dirty pages which happen to
2912                 * be hanging around before removing the file.  This
2913                 * shouldn't happen very often and mostly on file
2914                 * systems mounted "nocto".
2915                 */
2916                if (vn_has_cached_data(vp) &&
2917                    ((rp->r_flags & RDIRTY) || rp->r_count > 0)) {
2918                        error = nfs3_putpage(vp, (offset_t)0, 0, 0, cr, ct);
2919                        if (error && (error == ENOSPC || error == EDQUOT)) {
2920                                mutex_enter(&rp->r_statelock);
2921                                if (!rp->r_error)
2922                                        rp->r_error = error;
2923                                mutex_exit(&rp->r_statelock);
2924                        }
2925                }

2927                setdiropargs3(&args.object, nm, dvp);

2929                douprintf = 1;

2931                t = gethrtime();

2933                error = rfs3call(VTOMI(dvp), NFSPROC3_REMOVE,
2934                    xdr_diropargs3, (caddr_t)&args,
2935                    xdr_REMOVE3res, (caddr_t)&res, cr,
2936                    &douprintf, &res.status, 0, NULL);

2938                /*
2939                 * The xattr dir may be gone after last attr is removed,
2940                 * so flush it from dnlc.
2941                 */
2942                if (dvp->v_flag & V_XATTRDIR)
2943                        dnlc_purge_vp(dvp);

2945                PURGE_ATTRCACHE(vp);

2947                if (error) {
2948                        PURGE_ATTRCACHE(dvp);
2949                } else {
2950                        error = geterrno3(res.status);
2951                        if (!error) {
2952                                nfs3_cache_wcc_data(dvp, &res.resok.dir_wcc, t,
2953                                    cr);
2954                                if (HAVE_RDDIR_CACHE(drp))
2955                                        nfs_purge_rddir_cache(dvp);
2956                        } else {
2957                                nfs3_cache_wcc_data(dvp, &res.resfail.dir_wcc,
2958                                    t, cr);
2959                                PURGE_STALE_FH(error, dvp, cr);
2960                        }
2961                }
2962        }

2964        if (error == 0) {
2965                vnevent_remove(vp, dvp, nm, ct);
```



```
2966        }
2967        VN_RELE(vp);

2969        nfs_rw_exit(&drp->r_rwlock);

2971        return (error);
2972 }

2974 /* ARGSUSED */
2975 static int
2976 nfs3_link(vnode_t *tdvp, vnode_t *svp, char *tnm, cred_t *cr,
2977        caller_context_t *ct, int flags)
2978 {
2979        int error;
2980        LINK3args args;
2981        LINK3res res;
2982        vnode_t *realvp;
2983        int douprintf;
2984        mntinfo_t *mi;
2985        rnode_t *tdrp;
2986        hrtime_t t;

2988        if (nfs_zone() != VTOMI(tdvp)->mi_zone)
2989                return (EPERM);
2990        if (VOP_REALVP(svp, &realvp, ct) == 0)
2991                svp = realvp;

2993        mi = VTOMI(svp);

2995        if (!(mi->mi_flags & MI_LINK))
2996                return (EOPNOTSUPP);

2998        args.file = *VTOFH3(svp);
2999        setdiropargs3(&args.link, tnm, tdvp);

3001        tdrp = VTOR(tdvp);
3002        if (nfs_rw_enter_sig(&tdrp->r_rwlock, RW_WRITER, INTR(tdvp)))
3003                return (EINTR);

3005        dnlc_remove(tdvp, tnm);

3007        douprintf = 1;

3009        t = gethrtime();

3011        error = rfs3call(mi, NFSPROC3_LINK,
3012            xdr_LINK3args, (caddr_t)&args,
3013            xdr_LINK3res, (caddr_t)&res, cr,
3014            &douprintf, &res.status, 0, NULL);

3016        if (error) {
3017                PURGE_ATTRCACHE(tdvp);
3018                PURGE_ATTRCACHE(svp);
3019                nfs_rw_exit(&tdrp->r_rwlock);
3020                return (error);
3021        }

3023        error = geterrno3(res.status);

3025        if (!error) {
3026                nfs3_cache_post_op_attr(svp, &res.resok.file_attributes, t, cr);
3027                nfs3_cache_wcc_data(tdvp, &res.resok.linkdir_wcc, t, cr);
3028                if (HAVE_RDDIR_CACHE(tdrp))
3029                        nfs_purge_rddir_cache(tdvp);
3030                dnlc_update(tdvp, tnm, svp);
3031        } else {
```

```
3032                 nfs3_cache_post_op_attr(svp, &res.resfail.file_attributes, t,
3033                     cr);
3034                 nfs3_cache_wcc_data(tdvp, &res.resfail.linkdir_wcc, t, cr);
3035                 if (error == EOPNOTSUPP) {
3036                         mutex_enter(&mi->mi_lock);
3037                         mi->mi_flags &= ~MI_LINK;
3038                         mutex_exit(&mi->mi_lock);
3039                 }
3040         }

3042         nfs_rw_exit(&tdrp->r_rwlock);

3044         if (!error) {
3045                 /*
3046                  * Notify the source file of this link operation.
3047                  */
3048                 vnevent_link(svp, ct);
3049         }
3050         return (error);
3051 }

3053 /* ARGSUSED */
3054 static int
3055 nfs3_rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
3056         caller_context_t *ct, int flags)
3057 {
3058         vnode_t *realvp;

3060         if (nfs_zone() != VTOMI(odvp)->mi_zone)
3061                 return (EPERM);
3062         if (VOP_REALVP(ndvp, &realvp, ct) == 0)
3063                 ndvp = realvp;

3065         return (nfs3rename(odvp, onm, ndvp, nnm, cr, ct));
3066 }

3068 /*
3069  * nfs3rename does the real work of renaming in NFS Version 3.
3070  */
3071 static int
3072 nfs3rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
3073     caller_context_t *ct)
3074 {
3075         int error;
3076         RENAME3args args;
3077         RENAME3res res;
3078         int douprintf;
3079         vnode_t *nvp = NULL;
3080         vnode_t *ovp = NULL;
3081         char *tmpname;
3082         rnode_t *rp;
3083         rnode_t *odrp;
3084         rnode_t *ndrp;
3085         hrtime_t t;

3087         ASSERT(nfs_zone() == VTOMI(odvp)->mi_zone);

3089         if (strcmp(onm, ".") == 0 || strcmp(onm, "..") == 0 ||
3090             strcmp(nnm, ".") == 0 || strcmp(nnm, "..") == 0)
3091                 return (EINVAL);

3093         odrp = VTOR(odvp);
3094         ndrp = VTOR(ndvp);
3095         if ((intptr_t)odrp < (intptr_t)ndrp) {
3096                 if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR(odvp)))
3097                         return (EINTR);
```



```
3098                 if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR(ndvp))) {
3099                         nfs_rw_exit(&odrp->r_rwlock);
3100                         return (EINTR);
3101                 }
3102         } else {
3103                 if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR(ndvp)))
3104                         return (EINTR);
3105                 if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR(odvp))) {
3106                         nfs_rw_exit(&ndrp->r_rwlock);
3107                         return (EINTR);
3108                 }
3109         }

3111         /*
3112          * Lookup the target file.  If it exists, it needs to be
3113          * checked to see whether it is a mount point and whether
3114          * it is active (open).
3115          */
3116         error = nfs3lookup(ndvp, nnm, &nvp, NULL, 0, NULL, cr, 0);
3117         if (!error) {
3118                 /*
3119                  * If this file has been mounted on, then just
3120                  * return busy because renaming to it would remove
3121                  * the mounted file system from the name space.
3122                  */
3123                 if (vn_mountedvfs(nvp) != NULL) {
3124                         VN_RELE(nvp);
3125                         nfs_rw_exit(&odrp->r_rwlock);
3126                         nfs_rw_exit(&ndrp->r_rwlock);
3127                         return (EBUSY);
3128                 }

3130                 /*
3131                  * Purge the name cache of all references to this vnode
3132                  * so that we can check the reference count to infer
3133                  * whether it is active or not.
3134                  */
3135                 /*
3136                  * First just remove the entry from the name cache, as it
3137                  * is most likely the only entry for this vp.
3138                  */
3139                 dnlc_remove(ndvp, nnm);
3140                 /*
3141                  * If the file has a v_count > 1 then there may be more
3142                  * than one entry in the name cache due multiple links
3143                  * or an open file, but we don't have the real reference
3144                  * count so flush all possible entries.
3145                  */
3146                 if (nvp->v_count > 1)
3147                         dnlc_purge_vp(nvp);

3149                 /*
3150                  * If the vnode is active and is not a directory,
3151                  * arrange to rename it to a
3152                  * temporary file so that it will continue to be
3153                  * accessible.  This implements the "unlink-open-file"
3154                  * semantics for the target of a rename operation.
3155                  * Before doing this though, make sure that the
3156                  * source and target files are not already the same.
3157                  */
3158                 if (nvp->v_count > 1 && nvp->v_type != VDIR) {
3159                         /*
3160                          * Lookup the source name.
3161                          */
3162                         error = nfs3lookup(odvp, onm, &ovp, NULL, 0, NULL,
3163                             cr, 0);
```

```
3165                        /*
3166                         * The source name *should* already exist.
3167                         */
3168                        if (error) {
3169                                VN_RELE(nvp);
3170                                nfs_rw_exit(&odrp->r_rwlock);
3171                                nfs_rw_exit(&ndrp->r_rwlock);
3172                                return (error);
3173                        }

3175                        /*
3176                         * Compare the two vnodes.  If they are the same,
3177                         * just release all held vnodes and return success.
3178                         */
3179                        if (ovp == nvp) {
3180                                VN_RELE(ovp);
3181                                VN_RELE(nvp);
3182                                nfs_rw_exit(&odrp->r_rwlock);
3183                                nfs_rw_exit(&ndrp->r_rwlock);
3184                                return (0);
3185                        }

3187                        /*
3188                         * Can't mix and match directories and non-
3189                         * directories in rename operations.  We already
3190                         * know that the target is not a directory.  If
3191                         * the source is a directory, return an error.
3192                         */
3193                        if (ovp->v_type == VDIR) {
3194                                VN_RELE(ovp);
3195                                VN_RELE(nvp);
3196                                nfs_rw_exit(&odrp->r_rwlock);
3197                                nfs_rw_exit(&ndrp->r_rwlock);
3198                                return (ENOTDIR);
3199                        }

3201                        /*
3202                         * The target file exists, is not the same as
3203                         * the source file, and is active.  Link it
3204                         * to a temporary filename to avoid having
3205                         * the server removing the file completely.
3206                         */
3207                        tmpname = newname();
3208                        error = nfs3_link(ndvp, nvp, tmpname, cr, NULL, 0);
3209                        if (error == EOPNOTSUPP) {
3210                                error = nfs3_rename(ndvp, nnm, ndvp, tmpname,
3211                                    cr, NULL, 0);
3212                        }
3213                        if (error) {
3214                                kmem_free(tmpname, MAXNAMELEN);
3215                                VN_RELE(ovp);
3216                                VN_RELE(nvp);
3217                                nfs_rw_exit(&odrp->r_rwlock);
3218                                nfs_rw_exit(&ndrp->r_rwlock);
3219                                return (error);
3220                        }
3221                        rp = VTOR(nvp);
3222                        mutex_enter(&rp->r_statelock);
3223                        if (rp->r_unldvp == NULL) {
3224                                VN_HOLD(ndvp);
3225                                rp->r_unldvp = ndvp;
3226                                if (rp->r_unlcred != NULL)
3227                                        crfree(rp->r_unlcred);
3228                                crhold(cr);
3229                                rp->r_unlcred = cr;
```



```
3230                                rp->r_unlname = tmpname;
3231                        } else {
3232                                kmem_free(rp->r_unlname, MAXNAMELEN);
3233                                rp->r_unlname = tmpname;
3234                        }
3235                        mutex_exit(&rp->r_statelock);
3236                }
3237        }

3239        if (ovp == NULL) {
3240                /*
3241                 * When renaming directories to be a subdirectory of a
3242                 * different parent, the dnlc entry for ".." will no
3243                 * longer be valid, so it must be removed.
3244                 *
3245                 * We do a lookup here to determine whether we are renaming
3246                 * a directory and we need to check if we are renaming
3247                 * an unlinked file.  This might have already been done
3248                 * in previous code, so we check ovp == NULL to avoid
3249                 * doing it twice.
3250                 */

3252                error = nfs3lookup(odvp, onm, &ovp, NULL, 0, NULL, cr, 0);
3253                /*
3254                 * The source name *should* already exist.
3255                 */
3256                if (error) {
3257                        nfs_rw_exit(&odrp->r_rwlock);
3258                        nfs_rw_exit(&ndrp->r_rwlock);
3259                        if (nvp) {
3260                                VN_RELE(nvp);
3261                        }
3262                        return (error);
3263                }
3264                ASSERT(ovp != NULL);
3265        }

3267        dnlc_remove(odvp, onm);
3268        dnlc_remove(ndvp, nnm);

3270        setdiropargs3(&args.from, onm, odvp);
3271        setdiropargs3(&args.to, nnm, ndvp);

3273        douprintf = 1;

3275        t = gethrtime();

3277        error = rfs3call(VTOMI(odvp), NFSPROC3_RENAME,
3278            xdr_RENAME3args, (caddr_t)&args,
3279            xdr_RENAME3res, (caddr_t)&res, cr,
3280            &douprintf, &res.status, 0, NULL);

3282        if (error) {
3283                PURGE_ATTRCACHE(odvp);
3284                PURGE_ATTRCACHE(ndvp);
3285                VN_RELE(ovp);
3286                nfs_rw_exit(&odrp->r_rwlock);
3287                nfs_rw_exit(&ndrp->r_rwlock);
3288                if (nvp) {
3289                        VN_RELE(nvp);
3290                }
3291                return (error);
3292        }

3294        error = geterrno3(res.status);
```

```
3296         if (!error) {
3297                 nfs3_cache_wcc_data(odvp, &res.resok.fromdir_wcc, t, cr);
3298                 if (HAVE_RDDIR_CACHE(odrp))
3299                         nfs_purge_rddir_cache(odvp);
3300                 if (ndvp != odvp) {
3301                         nfs3_cache_wcc_data(ndvp, &res.resok.todir_wcc, t, cr);
3302                         if (HAVE_RDDIR_CACHE(ndrp))
3303                                 nfs_purge_rddir_cache(ndvp);
3304                 }
3305                 /*
3306                  * when renaming directories to be a subdirectory of a
3307                  * different parent, the dnlc entry for ".." will no
3308                  * longer be valid, so it must be removed
3309                  */
3310                 rp = VTOR(ovp);
3311                 if (ndvp != odvp) {
3312                         if (ovp->v_type == VDIR) {
3313                                 dnlc_remove(ovp, "..");
3314                                 if (HAVE_RDDIR_CACHE(rp))
3315                                         nfs_purge_rddir_cache(ovp);
3316                         }
3317                 }

3319                 /*
3320                  * If we are renaming the unlinked file, update the
3321                  * r_unldvp and r_unlname as needed.
3322                  */
3323                 mutex_enter(&rp->r_statelock);
3324                 if (rp->r_unldvp != NULL) {
3325                         if (strcmp(rp->r_unlname, onm) == 0) {
3326                                 (void) strncpy(rp->r_unlname, nnm, MAXNAMELEN);
3327                                 rp->r_unlname[MAXNAMELEN - 1] = '\0';

3329                                 if (ndvp != rp->r_unldvp) {
3330                                         VN_RELE(rp->r_unldvp);
3331                                         rp->r_unldvp = ndvp;
3332                                         VN_HOLD(ndvp);
3333                                 }
3334                         }
3335                 }
3336                 mutex_exit(&rp->r_statelock);
3337         } else {
3338                 nfs3_cache_wcc_data(odvp, &res.resfail.fromdir_wcc, t, cr);
3339                 if (ndvp != odvp) {
3340                         nfs3_cache_wcc_data(ndvp, &res.resfail.todir_wcc, t,
3341                             cr);
3342                 }
3343                 /*
3344                  * System V defines rename to return EEXIST, not
3345                  * ENOTEMPTY if the target directory is not empty.
3346                  * Over the wire, the error is NFSERR_ENOTEMPTY
3347                  * which geterrno maps to ENOTEMPTY.
3348                  */
3349                 if (error == ENOTEMPTY)
3350                         error = EEXIST;
3351         }

3353         if (error == 0) {
3354                 if (nvp)
3355                         vnevent_rename_dest(nvp, ndvp, nnm, ct);

3357                 if (odvp != ndvp)
3358                         vnevent_rename_dest_dir(ndvp, ct);
3359                 ASSERT(ovp != NULL);
3360                 vnevent_rename_src(ovp, odvp, onm, ct);
3361         }
```



```
3363         if (nvp) {
3364                 VN_RELE(nvp);
3365         }
3366         VN_RELE(ovp);

3368         nfs_rw_exit(&odrp->r_rwlock);
3369         nfs_rw_exit(&ndrp->r_rwlock);

3371         return (error);
3372 }

3374 /* ARGSUSED */
3375 static int
3376 nfs3_mkdir(vnode_t *dvp, char *nm, struct vattr *va, vnode_t **vpp, cred_t *cr,
3377         caller_context_t *ct, int flags, vsecattr_t *vsecp)
3378 {
3379         int error;
3380         MKDIR3args args;
3381         MKDIR3res res;
3382         int douprintf;
3383         struct vattr vattr;
3384         vnode_t *vp;
3385         rnode_t *drp;
3386         hrtime_t t;

3388         if (nfs_zone() != VTOMI(dvp)->mi_zone)
3389                 return (EPERM);
3390         setdiropargs3(&args.where, nm, dvp);

3392         /*
3393          * Decide what the group-id and set-gid bit of the created directory
3394          * should be.  May have to do a setattr to get the gid right.
3395          */
3396         error = setdirgid(dvp, &va->va_gid, cr);
3397         if (error)
3398                 return (error);
3399         error = setdirmode(dvp, &va->va_mode, cr);
3400         if (error)
3401                 return (error);
3402         va->va_mask |= AT_MODE|AT_GID;

3404         error = vattr_to_sattr3(va, &args.attributes);
3405         if (error) {
3406                 /* req time field(s) overflow - return immediately */
3407                 return (error);
3408         }

3410         drp = VTOR(dvp);
3411         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
3412                 return (EINTR);

3414         dnlc_remove(dvp, nm);

3416         douprintf = 1;

3418         t = gethrtime();

3420         error = rfs3call(VTOMI(dvp), NFSPROC3_MKDIR,
3421             xdr_MKDIR3args, (caddr_t)&args,
3422             xdr_MKDIR3res, (caddr_t)&res, cr,
3423             &douprintf, &res.status, 0, NULL);

3425         if (error) {
3426                 PURGE_ATTRCACHE(dvp);
3427                 nfs_rw_exit(&drp->r_rwlock);
```

```
3428                return (error);
3429        }

3431        error = geterrno3(res.status);
3432        if (!error) {
3433                nfs3_cache_wcc_data(dvp, &res.resok.dir_wcc, t, cr);
3434                if (HAVE_RDDIR_CACHE(drp))
3435                        nfs_purge_rddir_cache(dvp);

3437                if (!res.resok.obj.handle_follows) {
3438                        error = nfs3lookup(dvp, nm, &vp, NULL, 0, NULL, cr, 0);
3439                        if (error) {
3440                                nfs_rw_exit(&drp->r_rwlock);
3441                                return (error);
3442                        }
3443                } else {
3444                        if (res.resok.obj_attributes.attributes) {
3445                                vp = makenfs3node(&res.resok.obj.handle,
3446                                    &res.resok.obj_attributes.attr,
3447                                    dvp->v_vfsp, t, cr, NULL, NULL);
3448                        } else {
3449                                vp = makenfs3node(&res.resok.obj.handle, NULL,
3450                                    dvp->v_vfsp, t, cr, NULL, NULL);
3451                                if (vp->v_type == VNON) {
3452                                        vattr.va_mask = AT_TYPE;
3453                                        error = nfs3getattr(vp, &vattr, cr);
3454                                        if (error) {
3455                                                VN_RELE(vp);
3456                                                nfs_rw_exit(&drp->r_rwlock);
3457                                                return (error);
3458                                        }
3459                                        vp->v_type = vattr.va_type;
3460                                }
3461                        }
3462                        dnlc_update(dvp, nm, vp);
3463                }
3464                if (va->va_gid != VTOR(vp)->r_attr.va_gid) {
3465                        va->va_mask = AT_GID;
3466                        (void) nfs3setattr(vp, va, 0, cr);
3467                }
3468                *vpp = vp;
3469        } else {
3470                nfs3_cache_wcc_data(dvp, &res.resfail.dir_wcc, t, cr);
3471                PURGE_STALE_FH(error, dvp, cr);
3472        }

3474        nfs_rw_exit(&drp->r_rwlock);

3476        return (error);
3477 }

3479 /* ARGSUSED */
3480 static int
3481 nfs3_rmdir(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
3482        caller_context_t *ct, int flags)
3483 {
3484        int error;
3485        RMDIR3args args;
3486        RMDIR3res res;
3487        vnode_t *vp;
3488        int douprintf;
3489        rnode_t *drp;
3490        hrtime_t t;

3492        if (nfs_zone() != VTOMI(dvp)->mi_zone)
3493                return (EPERM);
```



```
3494        drp = VTOR(dvp);
3495        if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
3496                return (EINTR);

3498        /*
3499         * Attempt to prevent a rmdir(".") from succeeding.
3500         */
3501        error = nfs3lookup(dvp, nm, &vp, NULL, 0, NULL, cr, 0);
3502        if (error) {
3503                nfs_rw_exit(&drp->r_rwlock);
3504                return (error);
3505        }

3507        if (vp == cdir) {
3508                VN_RELE(vp);
3509                nfs_rw_exit(&drp->r_rwlock);
3510                return (EINVAL);
3511        }

3513        setdiropargs3(&args.object, nm, dvp);

3515        /*
3516         * First just remove the entry from the name cache, as it
3517         * is most likely an entry for this vp.
3518         */
3519        dnlc_remove(dvp, nm);

3521        /*
3522         * If there vnode reference count is greater than one, then
3523         * there may be additional references in the DNLC which will
3524         * need to be purged.  First, trying removing the entry for
3525         * the parent directory and see if that removes the additional
3526         * reference(s).  If that doesn't do it, then use dnlc_purge_vp
3527         * to completely remove any references to the directory which
3528         * might still exist in the DNLC.
3529         */
3530        if (vp->v_count > 1) {
3531                dnlc_remove(vp, "..");
3532                if (vp->v_count > 1)
3533                        dnlc_purge_vp(vp);
3534        }

3536        douprintf = 1;

3538        t = gethrtime();

3540        error = rfs3call(VTOMI(dvp), NFSPROC3_RMDIR,
3541            xdr_diropargs3, (caddr_t)&args,
3542            xdr_RMDIR3res, (caddr_t)&res, cr,
3543            &douprintf, &res.status, 0, NULL);

3545        PURGE_ATTRCACHE(vp);

3547        if (error) {
3548                PURGE_ATTRCACHE(dvp);
3549                VN_RELE(vp);
3550                nfs_rw_exit(&drp->r_rwlock);
3551                return (error);
3552        }

3554        error = geterrno3(res.status);
3555        if (!error) {
3556                nfs3_cache_wcc_data(dvp, &res.resok.dir_wcc, t, cr);
3557                if (HAVE_RDDIR_CACHE(drp))
3558                        nfs_purge_rddir_cache(dvp);
3559                if (HAVE_RDDIR_CACHE(VTOR(vp)))
```

```
3560                        nfs_purge_rddir_cache(vp);
3561        } else {
3562                nfs3_cache_wcc_data(dvp, &res.resfail.dir_wcc, t, cr);
3563                PURGE_STALE_FH(error, dvp, cr);
3564                /*
3565                 * System V defines rmdir to return EEXIST, not
3566                 * ENOTEMPTY if the directory is not empty.  Over
3567                 * the wire, the error is NFSERR_ENOTEMPTY which
3568                 * geterrno maps to ENOTEMPTY.
3569                 */
3570                if (error == ENOTEMPTY)
3571                        error = EEXIST;
3572        }

3574        if (error == 0) {
3575                vnevent_rmdir(vp, dvp, nm, ct);
3576        }
3577        VN_RELE(vp);

3579        nfs_rw_exit(&drp->r_rwlock);

3581        return (error);
3582 }

3584 /* ARGSUSED */
3585 static int
3586 nfs3_symlink(vnode_t *dvp, char *lnm, struct vattr *tva, char *tnm, cred_t *cr,
3587        caller_context_t *ct, int flags)
3588 {
3589        int error;
3590        SYMLINK3args args;
3591        SYMLINK3res res;
3592        int douprintf;
3593        mntinfo_t *mi;
3594        vnode_t *vp;
3595        rnode_t *rp;
3596        char *contents;
3597        rnode_t *drp;
3598        hrtime_t t;

3600        mi = VTOMI(dvp);

3602        if (nfs_zone() != mi->mi_zone)
3603                return (EPERM);
3604        if (!(mi->mi_flags & MI_SYMLINK))
3605                return (EOPNOTSUPP);

3607        setdiropargs3(&args.where, lnm, dvp);
3608        error = vattr_to_sattr3(tva, &args.symlink.symlink_attributes);
3609        if (error) {
3610                /* req time field(s) overflow - return immediately */
3611                return (error);
3612        }
3613        args.symlink.symlink_data = tnm;

3615        drp = VTOR(dvp);
3616        if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
3617                return (EINTR);

3619        dnlc_remove(dvp, lnm);

3621        douprintf = 1;

3623        t = gethrtime();

3625        error = rfs3call(mi, NFSPROC3_SYMLINK,
```



```
3626            xdr_SYMLINK3args, (caddr_t)&args,
3627            xdr_SYMLINK3res, (caddr_t)&res, cr,
3628            &douprintf, &res.status, 0, NULL);

3630        if (error) {
3631                PURGE_ATTRCACHE(dvp);
3632                nfs_rw_exit(&drp->r_rwlock);
3633                return (error);
3634        }

3636        error = geterrno3(res.status);
3637        if (!error) {
3638                nfs3_cache_wcc_data(dvp, &res.resok.dir_wcc, t, cr);
3639                if (HAVE_RDDIR_CACHE(drp))
3640                        nfs_purge_rddir_cache(dvp);

3642                if (res.resok.obj.handle_follows) {
3643                        if (res.resok.obj_attributes.attributes) {
3644                                vp = makenfs3node(&res.resok.obj.handle,
3645                                    &res.resok.obj_attributes.attr,
3646                                    dvp->v_vfsp, t, cr, NULL, NULL);
3647                        } else {
3648                                vp = makenfs3node(&res.resok.obj.handle, NULL,
3649                                    dvp->v_vfsp, t, cr, NULL, NULL);
3650                                vp->v_type = VLNK;
3651                                vp->v_rdev = 0;
3652                        }
3653                        dnlc_update(dvp, lnm, vp);
3654                        rp = VTOR(vp);
3655                        if (nfs3_do_symlink_cache &&
3656                            rp->r_symlink.contents == NULL) {

3658                                contents = kmem_alloc(MAXPATHLEN,
3659                                    KM_NOSLEEP);

3661                                if (contents != NULL) {
3662                                        mutex_enter(&rp->r_statelock);
3663                                        if (rp->r_symlink.contents == NULL) {
3664                                                rp->r_symlink.len = strlen(tnm);
3665                                                bcopy(tnm, contents,
3666                                                    rp->r_symlink.len);
3667                                                rp->r_symlink.contents =
3668                                                    contents;
3669                                                rp->r_symlink.size = MAXPATHLEN;
3670                                                mutex_exit(&rp->r_statelock);
3671                                        } else {
3672                                                mutex_exit(&rp->r_statelock);
3673                                                kmem_free((void *)contents,
3674                                                    MAXPATHLEN);
3675                                        }
3676                                }
3677                        }
3678                        VN_RELE(vp);
3679                }
3680        } else {
3681                nfs3_cache_wcc_data(dvp, &res.resfail.dir_wcc, t, cr);
3682                PURGE_STALE_FH(error, dvp, cr);
3683                if (error == EOPNOTSUPP) {
3684                        mutex_enter(&mi->mi_lock);
3685                        mi->mi_flags &= ~MI_SYMLINK;
3686                        mutex_exit(&mi->mi_lock);
3687                }
3688        }

3690        nfs_rw_exit(&drp->r_rwlock);
```

```
3692         return (error);
3693 }

3695 #ifdef DEBUG
3696 static int nfs3_readdir_cache_hits = 0;
3697 static int nfs3_readdir_cache_shorts = 0;
3698 static int nfs3_readdir_cache_waits = 0;
3699 static int nfs3_readdir_cache_misses = 0;
3700 static int nfs3_readdir_readahead = 0;
3701 #endif

3703 static int nfs3_shrinkreaddir = 0;

3705 /*
3706  * Read directory entries.
3707  * There are some weird things to look out for here.  The uio_loffset
3708  * field is either 0 or it is the offset returned from a previous
3709  * readdir.  It is an opaque value used by the server to find the
3710  * correct directory block to read. The count field is the number
3711  * of blocks to read on the server.  This is advisory only, the server
3712  * may return only one block's worth of entries.  Entries may be compressed
3713  * on the server.
3714  */
3715 /* ARGSUSED */
3716 static int
3717 nfs3_readdir(vnode_t *vp, struct uio *uiop, cred_t *cr, int *eofp,
3718         caller_context_t *ct, int flags)
3719 {
3720         int error;
3721         size_t count;
3722         rnode_t *rp;
3723         rddir_cache *rdc;
3724         rddir_cache *nrdc;
3725         rddir_cache *rrdc;
3726 #ifdef DEBUG
3727         int missed;
3728 #endif
3729         int doreadahead;
3730         rddir_cache srdc;
3731         avl_index_t where;

3733         if (nfs_zone() != VTOMI(vp)->mi_zone)
3734                 return (EIO);
3735         rp = VTOR(vp);

3737         ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));

3739         /*
3740          * Make sure that the directory cache is valid.
3741          */
3742         if (HAVE_RDDIR_CACHE(rp)) {
3743                 if (nfs_disable_rddir_cache) {
3744                         /*
3745                          * Setting nfs_disable_rddir_cache in /etc/system
3746                          * allows interoperability with servers that do not
3747                          * properly update the attributes of directories.
3748                          * Any cached information gets purged before an
3749                          * access is made to it.
3750                          */
3751                         nfs_purge_rddir_cache(vp);
3752                 } else {
3753                         error = nfs3_validate_caches(vp, cr);
3754                         if (error)
3755                                 return (error);
3756                 }
3757         }
```



```
3759         /*
3760          * It is possible that some servers may not be able to correctly
3761          * handle a large READDIR or READDIRPLUS request due to bugs in
3762          * their implementation.  In order to continue to interoperate
3763          * with them, this workaround is provided to limit the maximum
3764          * size of a READDIRPLUS request to 1024.  In any case, the request
3765          * size is limited to MAXBSIZE.
3766          */
3767         count = MIN(uiop->uio_iov->iov_len,
3768             nfs3_shrinkreaddir ? 1024 : MAXBSIZE);

3770         nrdc = NULL;
3771 #ifdef DEBUG
3772         missed = 0;
3773 #endif
3774 top:
3775         /*
3776          * Short circuit last readdir which always returns 0 bytes.
3777          * This can be done after the directory has been read through
3778          * completely at least once.  This will set r_direof which
3779          * can be used to find the value of the last cookie.
3780          */
3781         mutex_enter(&rp->r_statelock);
3782         if (rp->r_direof != NULL &&
3783             uiop->uio_loffset == rp->r_direof->nfs3_ncookie) {
3784                 mutex_exit(&rp->r_statelock);
3785 #ifdef DEBUG
3786                 nfs3_readdir_cache_shorts++;
3787 #endif
3788                 if (eofp)
3789                         *eofp = 1;
3790                 if (nrdc != NULL)
3791                         rddir_cache_rele(nrdc);
3792                 return (0);
3793         }
3794         /*
3795          * Look for a cache entry.  Cache entries are identified
3796          * by the NFS cookie value and the byte count requested.
3797          */
3798         srdc.nfs3_cookie = uiop->uio_loffset;
3799         srdc.buflen = count;
3800         rdc = avl_find(&rp->r_dir, &srdc, &where);
3801         if (rdc != NULL) {
3802                 rddir_cache_hold(rdc);
3803                 /*
3804                  * If the cache entry is in the process of being
3805                  * filled in, wait until this completes.  The
3806                  * RDDIRWAIT bit is set to indicate that someone
3807                  * is waiting and then the thread currently
3808                  * filling the entry is done, it should do a
3809                  * cv_broadcast to wakeup all of the threads
3810                  * waiting for it to finish.
3811                  */
3812                 if (rdc->flags & RDDIR) {
3813                         nfs_rw_exit(&rp->r_rwlock);
3814                         rdc->flags |= RDDIRWAIT;
3815 #ifdef DEBUG
3816                         nfs3_readdir_cache_waits++;
3817 #endif
3818                         if (!cv_wait_sig(&rdc->cv, &rp->r_statelock)) {
3819                                 /*
3820                                  * We got interrupted, probably
3821                                  * the user typed ^C or an alarm
3822                                  * fired.  We free the new entry
3823                                  * if we allocated one.
```

```
3824                                 */
3825                                mutex_exit(&rp->r_statelock);
3826                                (void) nfs_rw_enter_sig(&rp->r_rwlock,
3827                                    RW_READER, FALSE);
3828                                rddir_cache_rele(rdc);
3829                                if (nrdc != NULL)
3830                                        rddir_cache_rele(nrdc);
3831                                return (EINTR);
3832                        }
3833                        mutex_exit(&rp->r_statelock);
3834                        (void) nfs_rw_enter_sig(&rp->r_rwlock,
3835                            RW_READER, FALSE);
3836                        rddir_cache_rele(rdc);
3837                        goto top;
3838                }
3839                /*
3840                 * Check to see if a readdir is required to
3841                 * fill the entry.  If so, mark this entry
3842                 * as being filled, remove our reference,
3843                 * and branch to the code to fill the entry.
3844                 */
3845                if (rdc->flags & RDDIRREQ) {
3846                        rdc->flags &= ~RDDIRREQ;
3847                        rdc->flags |= RDDIR;
3848                        if (nrdc != NULL)
3849                                rddir_cache_rele(nrdc);
3850                        nrdc = rdc;
3851                        mutex_exit(&rp->r_statelock);
3852                        goto bottom;
3853                }
3854 #ifdef DEBUG
3855                if (!missed)
3856                        nfs3_readdir_cache_hits++;
3857 #endif
3858                /*
3859                 * If an error occurred while attempting
3860                 * to fill the cache entry, just return it.
3861                 */
3862                if (rdc->error) {
3863                        error = rdc->error;
3864                        mutex_exit(&rp->r_statelock);
3865                        rddir_cache_rele(rdc);
3866                        if (nrdc != NULL)
3867                                rddir_cache_rele(nrdc);
3868                        return (error);
3869                }

3871                /*
3872                 * The cache entry is complete and good,
3873                 * copyout the dirent structs to the calling
3874                 * thread.
3875                 */
3876                error = uiomove(rdc->entries, rdc->entlen, UIO_READ, uiop);

3878                /*
3879                 * If no error occurred during the copyout,
3880                 * update the offset in the uio struct to
3881                 * contain the value of the next cookie
3882                 * and set the eof value appropriately.
3883                 */
3884                if (!error) {
3885                        uiop->uio_loffset = rdc->nfs3_ncookie;
3886                        if (eofp)
3887                                *eofp = rdc->eof;
3888                }
```



```
3890                /*
3891                 * Decide whether to do readahead.
3892                 *
3893                 * Don't if have already read to the end of
3894                 * directory.  There is nothing more to read.
3895                 *
3896                 * Don't if the application is not doing
3897                 * lookups in the directory.  The readahead
3898                 * is only effective if the application can
3899                 * be doing work while an async thread is
3900                 * handling the over the wire request.
3901                 */
3902                if (rdc->eof) {
3903                        rp->r_direof = rdc;
3904                        doreadahead = FALSE;
3905                } else if (!(rp->r_flags & RLOOKUP))
3906                        doreadahead = FALSE;
3907                else
3908                        doreadahead = TRUE;

3910                if (!doreadahead) {
3911                        mutex_exit(&rp->r_statelock);
3912                        rddir_cache_rele(rdc);
3913                        if (nrdc != NULL)
3914                                rddir_cache_rele(nrdc);
3915                        return (error);
3916                }

3918                /*
3919                 * Check to see whether we found an entry
3920                 * for the readahead.  If so, we don't need
3921                 * to do anything further, so free the new
3922                 * entry if one was allocated.  Otherwise,
3923                 * allocate a new entry, add it to the cache,
3924                 * and then initiate an asynchronous readdir
3925                 * operation to fill it.
3926                 */
3927                srdc.nfs3_cookie = rdc->nfs3_ncookie;
3928                srdc.buflen = count;
3929                rrdc = avl_find(&rp->r_dir, &srdc, &where);
3930                if (rrdc != NULL) {
3931                        if (nrdc != NULL)
3932                                rddir_cache_rele(nrdc);
3933                } else {
3934                        if (nrdc != NULL)
3935                                rrdc = nrdc;
3936                        else {
3937                                rrdc = rddir_cache_alloc(KM_NOSLEEP);
3938                        }
3939                        if (rrdc != NULL) {
3940                                rrdc->nfs3_cookie = rdc->nfs3_ncookie;
3941                                rrdc->buflen = count;
3942                                avl_insert(&rp->r_dir, rrdc, where);
3943                                rddir_cache_hold(rrdc);
3944                                mutex_exit(&rp->r_statelock);
3945                                rddir_cache_rele(rdc);
3946 #ifdef DEBUG
3947                                nfs3_readdir_readahead++;
3948 #endif
3949                                nfs_async_readdir(vp, rrdc, cr, do_nfs3readdir);
3950                                return (error);
3951                        }
3952                }

3954                mutex_exit(&rp->r_statelock);
3955                rddir_cache_rele(rdc);
```

```
3956                    return (error);
3957            }

3959            /*
3960             * Didn't find an entry in the cache.  Construct a new empty
3961             * entry and link it into the cache.  Other processes attempting
3962             * to access this entry will need to wait until it is filled in.
3963             *
3964             * Since kmem_alloc may block, another pass through the cache
3965             * will need to be taken to make sure that another process
3966             * hasn't already added an entry to the cache for this request.
3967             */
3968            if (nrdc == NULL) {
3969                    mutex_exit(&rp->r_statelock);
3970                    nrdc = rddir_cache_alloc(KM_SLEEP);
3971                    nrdc->nfs3_cookie = uiop->uio_loffset;
3972                    nrdc->buflen = count;
3973                    goto top;
3974            }

3976            /*
3977             * Add this entry to the cache.
3978             */
3979            avl_insert(&rp->r_dir, nrdc, where);
3980            rddir_cache_hold(nrdc);
3981            mutex_exit(&rp->r_statelock);

3983    bottom:
3984    #ifdef DEBUG
3985            missed = 1;
3986            nfs3_readdir_cache_misses++;
3987    #endif
3988            /*
3989             * Do the readdir.  This routine decides whether to use
3990             * READDIR or READDIRPLUS.
3991             */
3992            error = do_nfs3readdir(vp, nrdc, cr);

3994            /*
3995             * If this operation failed, just return the error which occurred.
3996             */
3997            if (error != 0)
3998                    return (error);

4000            /*
4001             * Since the RPC operation will have taken sometime and blocked
4002             * this process, another pass through the cache will need to be
4003             * taken to find the correct cache entry.  It is possible that
4004             * the correct cache entry will not be there (although one was
4005             * added) because the directory changed during the RPC operation
4006             * and the readdir cache was flushed.  In this case, just start
4007             * over.  It is hoped that this will not happen too often... :-)
4008             */
4009            nrdc = NULL;
4010            goto top;
4011            /* NOTREACHED */
4012    }

4014    static int
4015    do_nfs3readdir(vnode_t *vp, rddir_cache *rdc, cred_t *cr)
4016    {
4017            int error;
4018            rnode_t *rp;
4019            mntinfo_t *mi;

4021            rp = VTOR(vp);
```



```
4022            mi = VTOMI(vp);
4023            ASSERT(nfs_zone() == mi->mi_zone);
4024            /*
4025             * Issue the proper request.
4026             *
4027             * If the server does not support READDIRPLUS, then use READDIR.
4028             *
4029             * Otherwise --
4030             * Issue a READDIRPLUS if reading to fill an empty cache or if
4031             * an application has performed a lookup in the directory which
4032             * required an over the wire lookup.  The use of READDIRPLUS
4033             * will help to (re)populate the DNLC.
4034             */
4035            if (!(mi->mi_flags & MI_READDIRONLY) &&
4036                (rp->r_flags & (RLOOKUP | RREADDIRPLUS))) {
4037                    if (rp->r_flags & RREADDIRPLUS) {
4038                            mutex_enter(&rp->r_statelock);
4039                            rp->r_flags &= ~RREADDIRPLUS;
4040                            mutex_exit(&rp->r_statelock);
4041                    }
4042                    nfs3readdirplus(vp, rdc, cr);
4043                    if (rdc->error == EOPNOTSUPP)
4044                            nfs3readdir(vp, rdc, cr);
4045            } else
4046                    nfs3readdir(vp, rdc, cr);

4048            mutex_enter(&rp->r_statelock);
4049            rdc->flags &= ~RDDIR;
4050            if (rdc->flags & RDDIRWAIT) {
4051                    rdc->flags &= ~RDDIRWAIT;
4052                    cv_broadcast(&rdc->cv);
4053            }
4054            error = rdc->error;
4055            if (error)
4056                    rdc->flags |= RDDIRREQ;
4057            mutex_exit(&rp->r_statelock);

4059            rddir_cache_rele(rdc);

4061            return (error);
4062    }

4064    static void
4065    nfs3readdir(vnode_t *vp, rddir_cache *rdc, cred_t *cr)
4066    {
4067            int error;
4068            READDIR3args args;
4069            READDIR3vres res;
4070            vattr_t dva;
4071            rnode_t *rp;
4072            int douprintf;
4073            failinfo_t fi, *fip = NULL;
4074            mntinfo_t *mi;
4075            hrtime_t t;

4077            rp = VTOR(vp);
4078            mi = VTOMI(vp);
4079            ASSERT(nfs_zone() == mi->mi_zone);

4081            args.dir = *RTOFH3(rp);
4082            args.cookie = (cookie3)rdc->nfs3_cookie;
4083            args.cookieverf = rp->r_cookieverf;
4084            args.count = rdc->buflen;

4086            /*
4087             * NFS client failover support
```

```
4088          * suppress failover unless we have a zero cookie
4089          */
4090         if (args.cookie == (cookie3) 0) {
4091                 fi.vp = vp;
4092                 fi.fhp = (caddr_t)&args.dir;
4093                 fi.copyproc = nfs3copyfh;
4094                 fi.lookupproc = nfs3lookup;
4095                 fi.xattrdirproc = acl_getxattrdir3;
4096                 fip = &fi;
4097         }

4099 #ifdef DEBUG
4100         rdc->entries = rddir_cache_buf_alloc(rdc->buflen, KM_SLEEP);
4101 #else
4102         rdc->entries = kmem_alloc(rdc->buflen, KM_SLEEP);
4103 #endif

4105         res.entries = (dirent64_t *)rdc->entries;
4106         res.entries_size = rdc->buflen;
4107         res.dir_attributes.fres.vap = &dva;
4108         res.dir_attributes.fres.vp = vp;
4109         res.loff = rdc->nfs3_cookie;

4111         douprintf = 1;

4113         if (mi->mi_io_kstats) {
4114                 mutex_enter(&mi->mi_lock);
4115                 kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
4116                 mutex_exit(&mi->mi_lock);
4117         }

4119         t = gethrtime();

4121         error = rfs3call(VTOMI(vp), NFSPROC3_READDIR,
4122             xdr_READDIR3args, (caddr_t)&args,
4123             xdr_READDIR3vres, (caddr_t)&res, cr,
4124             &douprintf, &res.status, 0, fip);

4126         if (mi->mi_io_kstats) {
4127                 mutex_enter(&mi->mi_lock);
4128                 kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
4129                 mutex_exit(&mi->mi_lock);
4130         }

4132         if (error)
4133                 goto err;

4135         nfs3_cache_post_op_vattr(vp, &res.dir_attributes, t, cr);

4137         error = geterrno3(res.status);
4138         if (error) {
4139                 PURGE_STALE_FH(error, vp, cr);
4140                 goto err;
4141         }

4143         if (mi->mi_io_kstats) {
4144                 mutex_enter(&mi->mi_lock);
4145                 KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
4146                 KSTAT_IO_PTR(mi->mi_io_kstats)->nread += res.size;
4147                 mutex_exit(&mi->mi_lock);
4148         }

4150         rdc->nfs3_ncookie = res.loff;
4151         rp->r_cookieverf = res.cookieverf;
4152         rdc->eof = res.eof ? 1 : 0;
4153         rdc->entlen = res.size;
```



```
4154         ASSERT(rdc->entlen <= rdc->buflen);
4155         rdc->error = 0;
4156         return;

4158 err:
4159         kmem_free(rdc->entries, rdc->buflen);
4160         rdc->entries = NULL;
4161         rdc->error = error;
4162 }

4164 /*
4165  * Read directory entries.
4166  * There are some weird things to look out for here.  The uio_loffset
4167  * field is either 0 or it is the offset returned from a previous
4168  * readdir.  It is an opaque value used by the server to find the
4169  * correct directory block to read. The count field is the number
4170  * of blocks to read on the server.  This is advisory only, the server
4171  * may return only one block's worth of entries.  Entries may be compressed
4172  * on the server.
4173  */
4174 static void
4175 nfs3readdirplus(vnode_t *vp, rddir_cache *rdc, cred_t *cr)
4176 {
4177         int error;
4178         READDIRPLUS3args args;
4179         READDIRPLUS3vres res;
4180         vattr_t dva;
4181         rnode_t *rp;
4182         mntinfo_t *mi;
4183         int douprintf;
4184         failinfo_t fi, *fip = NULL;

4186         rp = VTOR(vp);
4187         mi = VTOMI(vp);
4188         ASSERT(nfs_zone() == mi->mi_zone);

4190         args.dir = *RTOFH3(rp);
4191         args.cookie = (cookie3)rdc->nfs3_cookie;
4192         args.cookieverf = rp->r_cookieverf;
4193         args.dircount = rdc->buflen;
4194         args.maxcount = mi->mi_tsize;

4196         /*
4197          * NFS client failover support
4198          * suppress failover unless we have a zero cookie
4199          */
4200         if (args.cookie == (cookie3)0) {
4201                 fi.vp = vp;
4202                 fi.fhp = (caddr_t)&args.dir;
4203                 fi.copyproc = nfs3copyfh;
4204                 fi.lookupproc = nfs3lookup;
4205                 fi.xattrdirproc = acl_getxattrdir3;
4206                 fip = &fi;
4207         }

4209 #ifdef DEBUG
4210         rdc->entries = rddir_cache_buf_alloc(rdc->buflen, KM_SLEEP);
4211 #else
4212         rdc->entries = kmem_alloc(rdc->buflen, KM_SLEEP);
4213 #endif

4215         res.entries = (dirent64_t *)rdc->entries;
4216         res.entries_size = rdc->buflen;
4217         res.dir_attributes.fres.vap = &dva;
4218         res.dir_attributes.fres.vp = vp;
4219         res.loff = rdc->nfs3_cookie;
```

```
4220         res.credentials = cr;

4222         douprintf = 1;

4224         if (mi->mi_io_kstats) {
4225                 mutex_enter(&mi->mi_lock);
4226                 kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
4227                 mutex_exit(&mi->mi_lock);
4228         }

4230         res.time = gethrtime();

4232         error = rfs3call(mi, NFSPROC3_READDIRPLUS,
4233             xdr_READDIRPLUS3args, (caddr_t)&args,
4234             xdr_READDIRPLUS3vres, (caddr_t)&res, cr,
4235             &douprintf, &res.status, 0, fip);

4237         if (mi->mi_io_kstats) {
4238                 mutex_enter(&mi->mi_lock);
4239                 kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
4240                 mutex_exit(&mi->mi_lock);
4241         }

4243         if (error) {
4244                 goto err;
4245         }

4247         nfs3_cache_post_op_vattr(vp, &res.dir_attributes, res.time, cr);

4249         error = geterrno3(res.status);
4250         if (error) {
4251                 PURGE_STALE_FH(error, vp, cr);
4252                 if (error == EOPNOTSUPP) {
4253                         mutex_enter(&mi->mi_lock);
4254                         mi->mi_flags |= MI_READDIRONLY;
4255                         mutex_exit(&mi->mi_lock);
4256                 }
4257                 goto err;
4258         }

4260         if (mi->mi_io_kstats) {
4261                 mutex_enter(&mi->mi_lock);
4262                 KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
4263                 KSTAT_IO_PTR(mi->mi_io_kstats)->nread += res.size;
4264                 mutex_exit(&mi->mi_lock);
4265         }

4267         rdc->nfs3_ncookie = res.loff;
4268         rp->r_cookieverf = res.cookieverf;
4269         rdc->eof = res.eof ? 1 : 0;
4270         rdc->entlen = res.size;
4271         ASSERT(rdc->entlen <= rdc->buflen);
4272         rdc->error = 0;

4274         return;

4276 err:
4277         kmem_free(rdc->entries, rdc->buflen);
4278         rdc->entries = NULL;
4279         rdc->error = error;
4280 }

4282 #ifdef DEBUG
4283 static int nfs3_bio_do_stop = 0;
4284 #endif
```



```
4286 static int
4287 nfs3_bio(struct buf *bp, stable_how *stab_comm, cred_t *cr)
4288 {
4289         rnode_t *rp = VTOR(bp->b_vp);
4290         int count;
4291         int error;
4292         cred_t *cred;
4293         offset_t offset;

4295         ASSERT(nfs_zone() == VTOMI(bp->b_vp)->mi_zone);
4296         offset = ldbtob(bp->b_lblkno);

4298         DTRACE_IO1(start, struct buf *, bp);

4300         if (bp->b_flags & B_READ) {
4301                 mutex_enter(&rp->r_statelock);
4302                 if (rp->r_cred != NULL) {
4303                         cred = rp->r_cred;
4304                         crhold(cred);
4305                 } else {
4306                         rp->r_cred = cr;
4307                         crhold(cr);
4308                         cred = cr;
4309                         crhold(cred);
4310                 }
4311                 mutex_exit(&rp->r_statelock);
4312         read_again:
4313                 error = bp->b_error = nfs3read(bp->b_vp, bp->b_un.b_addr,
4314                     offset, bp->b_bcount, &bp->b_resid, cred);
4315                 crfree(cred);
4316                 if (!error) {
4317                         if (bp->b_resid) {
4318                                 /*
4319                                  * Didn't get it all because we hit EOF,
4320                                  * zero all the memory beyond the EOF.
4321                                  */
4322                                 /* bzero(rdaddr + */
4323                                 bzero(bp->b_un.b_addr +
4324                                     bp->b_bcount - bp->b_resid, bp->b_resid);
4325                         }
4326                         mutex_enter(&rp->r_statelock);
4327                         if (bp->b_resid == bp->b_bcount &&
4328                             offset >= rp->r_size) {
4329                                 /*
4330                                  * We didn't read anything at all as we are
4331                                  * past EOF.  Return an error indicator back
4332                                  * but don't destroy the pages (yet).
4333                                  */
4334                                 error = NFS_EOF;
4335                         }
4336                         mutex_exit(&rp->r_statelock);
4337                 } else if (error == EACCES) {
4338                         mutex_enter(&rp->r_statelock);
4339                         if (cred != cr) {
4340                                 if (rp->r_cred != NULL)
4341                                         crfree(rp->r_cred);
4342                                 rp->r_cred = cr;
4343                                 crhold(cr);
4344                                 cred = cr;
4345                                 crhold(cred);
4346                                 mutex_exit(&rp->r_statelock);
4347                                 goto read_again;
4348                         }
4349                         mutex_exit(&rp->r_statelock);
4350                 }
4351         } else {
```

```
4352                 if (!(rp->r_flags & RSTALE)) {
4353                         mutex_enter(&rp->r_statelock);
4354                         if (rp->r_cred != NULL) {
4355                                 cred = rp->r_cred;
4356                                 crhold(cred);
4357                         } else {
4358                                 rp->r_cred = cr;
4359                                 crhold(cr);
4360                                 cred = cr;
4361                                 crhold(cred);
4362                         }
4363                         mutex_exit(&rp->r_statelock);
4364                 write_again:
4365                         mutex_enter(&rp->r_statelock);
4366                         count = MIN(bp->b_bcount, rp->r_size - offset);
4367                         mutex_exit(&rp->r_statelock);
4368                         if (count < 0)
4369                                 cmn_err(CE_PANIC, "nfs3_bio: write count < 0");
4370 #ifdef DEBUG
4371                         if (count == 0) {
4372                                 zcmn_err(getzoneid(), CE_WARN,
4373                                     "nfs3_bio: zero length write at %lld",
4374                                     offset);
4375                                 nfs_printfhandle(&rp->r_fh);
4376                                 if (nfs3_bio_do_stop)
4377                                         debug_enter("nfs3_bio");
4378                         }
4379 #endif
4380                         error = nfs3write(bp->b_vp, bp->b_un.b_addr, offset,
4381                             count, cred, stab_comm);
4382                         if (error == EACCES) {
4383                                 mutex_enter(&rp->r_statelock);
4384                                 if (cred != cr) {
4385                                         if (rp->r_cred != NULL)
4386                                                 crfree(rp->r_cred);
4387                                         rp->r_cred = cr;
4388                                         crhold(cr);
4389                                         crfree(cred);
4390                                         cred = cr;
4391                                         crhold(cred);
4392                                         mutex_exit(&rp->r_statelock);
4393                                         goto write_again;
4394                                 }
4395                                 mutex_exit(&rp->r_statelock);
4396                         }
4397                         bp->b_error = error;
4398                         if (error && error != EINTR) {
4399                                 /*
4400                                  * Don't print EDQUOT errors on the console.
4401                                  * Don't print asynchronous EACCES errors.
4402                                  * Don't print EFBIG errors.
4403                                  * Print all other write errors.
4404                                  */
4405                                 if (error != EDQUOT && error != EFBIG &&
4406                                     (error != EACCES ||
4407                                     !(bp->b_flags & B_ASYNC)))
4408                                         nfs_write_error(bp->b_vp, error, cred);
4409                                 /*
4410                                  * Update r_error and r_flags as appropriate.
4411                                  * If the error was ESTALE, then mark the
4412                                  * rnode as not being writeable and save
4413                                  * the error status.  Otherwise, save any
4414                                  * errors which occur from asynchronous
4415                                  * page invalidations.  Any errors occurring
4416                                  * from other operations should be saved
4417                                  * by the caller.
```



```
4418                                  */
4419                                 mutex_enter(&rp->r_statelock);
4420                                 if (error == ESTALE) {
4421                                         rp->r_flags |= RSTALE;
4422                                         if (!rp->r_error)
4423                                                 rp->r_error = error;
4424                                 } else if (!rp->r_error &&
4425                                     (bp->b_flags &
4426                                     (B_INVAL|B_FORCE|B_ASYNC)) ==
4427                                     (B_INVAL|B_FORCE|B_ASYNC)) {
4428                                         rp->r_error = error;
4429                                 }
4430                                 mutex_exit(&rp->r_statelock);
4431                         }
4432                         crfree(cred);
4433                 } else {
4434                         error = rp->r_error;
4435                         /*
4436                          * A close may have cleared r_error, if so,
4437                          * propagate ESTALE error return properly
4438                          */
4439                         if (error == 0)
4440                                 error = ESTALE;
4441                 }
4442         }
4443
4444         if (error != 0 && error != NFS_EOF)
4445                 bp->b_flags |= B_ERROR;
4446
4447         DTRACE_IO1(done, struct buf *, bp);
4448
4449         return (error);
4450 }
4451
4452 /* ARGSUSED */
4453 static int
4454 nfs3_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
4455 {
4456         rnode_t *rp;
4457
4458         if (nfs_zone() != VTOMI(vp)->mi_zone)
4459                 return (EIO);
4460         rp = VTOR(vp);
4461
4462         if (fidp->fid_len < (ushort_t)rp->r_fh.fh_len) {
4463                 fidp->fid_len = rp->r_fh.fh_len;
4464                 return (ENOSPC);
4465         }
4466         fidp->fid_len = rp->r_fh.fh_len;
4467         bcopy(rp->r_fh.fh_buf, fidp->fid_data, fidp->fid_len);
4468         return (0);
4469 }
4470
4471 /* ARGSUSED2 */
4472 static int
4473 nfs3_rwlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
4474 {
4475         rnode_t *rp = VTOR(vp);
4476
4477         if (!write_lock) {
4478                 (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
4479                 return (V_WRITELOCK_FALSE);
4480         }
4481
4482         if ((rp->r_flags & RDIRECTIO) || (VTOMI(vp)->mi_flags & MI_DIRECTIO)) {
4483                 (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
```

```
4484                 if (rp->r_mapcnt == 0 && !vn_has_cached_data(vp))
4485                         return (V_WRITELOCK_FALSE);
4486                 nfs_rw_exit(&rp->r_rwlock);
4487         }
4488
4489         (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, FALSE);
4490         return (V_WRITELOCK_TRUE);
4491 }
4492
4493 /* ARGSUSED */
4494 static void
4495 nfs3_rwunlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
4496 {
4497         rnode_t *rp = VTOR(vp);
4498
4499         nfs_rw_exit(&rp->r_rwlock);
4500 }
4501
4502 /* ARGSUSED */
4503 static int
4504 nfs3_seek(vnode_t *vp, offset_t ooff, offset_t *noffp, caller_context_t *ct)
4505 {
4506
4507         /*
4508          * Because we stuff the readdir cookie into the offset field
4509          * someone may attempt to do an lseek with the cookie which
4510          * we want to succeed.
4511          */
4512         if (vp->v_type == VDIR)
4513                 return (0);
4514         if (*noffp < 0)
4515                 return (EINVAL);
4516         return (0);
4517 }
4518
4519 /*
4520  * number of nfs3_bsize blocks to read ahead.
4521  */
4522 static int nfs3_nra = 4;
4523
4524 #ifdef DEBUG
4525 static int nfs3_lostpage = 0;   /* number of times we lost original page */
4526 #endif
4527
4528 /*
4529  * Return all the pages from [off..off+len) in file
4530  */
4531 /* ARGSUSED */
4532 static int
4533 nfs3_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
4534         page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
4535         enum seg_rw rw, cred_t *cr, caller_context_t *ct)
4536 {
4537         rnode_t *rp;
4538         int error;
4539         mntinfo_t *mi;
4540
4541         if (vp->v_flag & VNOMAP)
4542                 return (ENOSYS);
4543
4544         if (nfs_zone() != VTOMI(vp)->mi_zone)
4545                 return (EIO);
4546         if (protp != NULL)
4547                 *protp = PROT_ALL;
4548
4549         /*
```



```
4550          * Now valididate that the caches are up to date.
4551          */
4552         error = nfs3_validate_caches(vp, cr);
4553         if (error)
4554                 return (error);
4555
4556         rp = VTOR(vp);
4557         mi = VTOMI(vp);
4558 retry:
4559         mutex_enter(&rp->r_statelock);
4560
4561         /*
4562          * Don't create dirty pages faster than they
4563          * can be cleaned so that the system doesn't
4564          * get imbalanced.  If the async queue is
4565          * maxed out, then wait for it to drain before
4566          * creating more dirty pages.  Also, wait for
4567          * any threads doing pagewalks in the vop_getattr
4568          * entry points so that they don't block for
4569          * long periods.
4570          */
4571         if (rw == S_CREATE) {
4572                 while ((mi->mi_max_threads != 0 &&
4573                     rp->r_awcount > 2 * mi->mi_max_threads) ||
4574                     rp->r_gcount > 0)
4575                         cv_wait(&rp->r_cv, &rp->r_statelock);
4576         }
4577
4578         /*
4579          * If we are getting called as a side effect of an nfs_write()
4580          * operation the local file size might not be extended yet.
4581          * In this case we want to be able to return pages of zeroes.
4582          */
4583         if (off + len > rp->r_size + PAGEOFFSET && seg != segkmap) {
4584                 mutex_exit(&rp->r_statelock);
4585                 return (EFAULT);                /* beyond EOF */
4586         }
4587
4588         mutex_exit(&rp->r_statelock);
4589
  33         if (len <= PAGESIZE) {
  34                 error = nfs3_getapage(vp, off, len, protp, pl, plsz,
  35                     seg, addr, rw, cr);
  36         } else {
4590         error = pvn_getpages(nfs3_getapage, vp, off, len, protp,
4591             pl, plsz, seg, addr, rw, cr);
  39         }
4592
4593         switch (error) {
4594         case NFS_EOF:
4595                 nfs_purge_caches(vp, NFS_NOPURGE_DNLC, cr);
4596                 goto retry;
4597         case ESTALE:
4598                 PURGE_STALE_FH(error, vp, cr);
4599         }
4600
4601         return (error);
4602 }
4603
4604 /*
4605  * Called from pvn_getpages to get a particular page.
  53  * Called from pvn_getpages or nfs3_getpage to get a particular page.
4606  */
4607 /* ARGSUSED */
4608 static int
4609 nfs3_getapage(vnode_t *vp, u_offset_t off, size_t len, uint_t *protp,
```

```
4610         page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
4611         enum seg_rw rw, cred_t *cr)
4612 {
4613         rnode_t *rp;
4614         uint_t bsize;
4615         struct buf *bp;
4616         page_t *pp;
4617         u_offset_t lbn;
4618         u_offset_t io_off;
4619         u_offset_t blkoff;
4620         u_offset_t rablkoff;
4621         size_t io_len;
4622         uint_t blksize;
4623         int error;
4624         int readahead;
4625         int readahead_issued = 0;
4626         int ra_window; /* readahead window */
4627         page_t *pagefound;
4628         page_t *savepp;

4630         if (nfs_zone() != VTOMI(vp)->mi_zone)
4631                 return (EIO);
4632         rp = VTOR(vp);
4633         bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);

4635 reread:
4636         bp = NULL;
4637         pp = NULL;
4638         pagefound = NULL;

4640         if (pl != NULL)
4641                 pl[0] = NULL;

4643         error = 0;
4644         lbn = off / bsize;
4645         blkoff = lbn * bsize;

4647         /*
4648          * Queueing up the readahead before doing the synchronous read
4649          * results in a significant increase in read throughput because
4650          * of the increased parallelism between the async threads and
4651          * the process context.
4652          */
4653         if ((off & ((vp->v_vfsp->vfs_bsize) - 1)) == 0 &&
4654             rw != S_CREATE &&
4655             !(vp->v_flag & VNOCACHE)) {
4656                 mutex_enter(&rp->r_statelock);

4658                 /*
4659                  * Calculate the number of readaheads to do.
4660                  * a) No readaheads at offset = 0.
4661                  * b) Do maximum(nfs3_nra) readaheads when the readahead
4662                  *    window is closed.
4663                  * c) Do readaheads between 1 to (nfs3_nra - 1) depending
4664                  *    upon how far the readahead window is open or close.
4665                  * d) No readaheads if rp->r_nextr is not within the scope
4666                  *    of the readahead window (random i/o).
4667                  */

4669                 if (off == 0)
4670                         readahead = 0;
4671                 else if (blkoff == rp->r_nextr)
4672                         readahead = nfs3_nra;
4673                 else if (rp->r_nextr > blkoff &&
4674                     ((ra_window = (rp->r_nextr - blkoff) / bsize)
4675                     <= (nfs3_nra - 1)))
```



```
4676                         readahead = nfs3_nra - ra_window;
4677                 else
4678                         readahead = 0;

4680                 rablkoff = rp->r_nextr;
4681                 while (readahead > 0 && rablkoff + bsize < rp->r_size) {
4682                         mutex_exit(&rp->r_statelock);
4683                         if (nfs_async_readahead(vp, rablkoff + bsize,
4684                             addr + (rablkoff + bsize - off), seg, cr,
4685                             nfs3_readahead) < 0) {
4686                                 mutex_enter(&rp->r_statelock);
4687                                 break;
4688                         }
4689                         readahead--;
4690                         rablkoff += bsize;
4691                         /*
4692                          * Indicate that we did a readahead so
4693                          * readahead offset is not updated
4694                          * by the synchronous read below.
4695                          */
4696                         readahead_issued = 1;
4697                         mutex_enter(&rp->r_statelock);
4698                         /*
4699                          * set readahead offset to
4700                          * offset of last async readahead
4701                          * request.
4702                          */
4703                         rp->r_nextr = rablkoff;
4704                 }
4705                 mutex_exit(&rp->r_statelock);
4706         }

4708 again:
4709         if ((pagefound = page_exists(vp, off)) == NULL) {
4710                 if (pl == NULL) {
4711                         (void) nfs_async_readahead(vp, blkoff, addr, seg, cr,
4712                             nfs3_readahead);
4713                 } else if (rw == S_CREATE) {
4714                         /*
4715                          * Block for this page is not allocated, or the offset
4716                          * is beyond the current allocation size, or we're
4717                          * allocating a swap slot and the page was not found,
4718                          * so allocate it and return a zero page.
4719                          */
4720                         if ((pp = page_create_va(vp, off,
4721                             PAGESIZE, PG_WAIT, seg, addr)) == NULL)
4722                                 cmn_err(CE_PANIC, "nfs3_getapage: page_create");
4723                         io_len = PAGESIZE;
4724                         mutex_enter(&rp->r_statelock);
4725                         rp->r_nextr = off + PAGESIZE;
4726                         mutex_exit(&rp->r_statelock);
4727                 } else {
4728                         /*
4729                          * Need to go to server to get a BLOCK, exception to
4730                          * that being while reading at offset = 0 or doing
4731                          * random i/o, in that case read only a PAGE.
4732                          */
4733                         mutex_enter(&rp->r_statelock);
4734                         if (blkoff < rp->r_size &&
4735                             blkoff + bsize >= rp->r_size) {
4736                                 /*
4737                                  * If only a block or less is left in
4738                                  * the file, read all that is remaining.
4739                                  */
4740                                 if (rp->r_size <= off) {
4741                                         /*
```

```
4742                                     * Trying to access beyond EOF,
4743                                     * set up to get at least one page.
4744                                     */
4745                                    blksize = off + PAGESIZE - blkoff;
4746                            } else
4747                                    blksize = rp->r_size - blkoff;
4748                    } else if ((off == 0) ||
4749                        (off != rp->r_nextr && !readahead_issued)) {
4750                            blksize = PAGESIZE;
4751                            blkoff = off; /* block = page here */
4752                    } else
4753                            blksize = bsize;
4754                    mutex_exit(&rp->r_statelock);

4756                    pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
4757                        &io_len, blkoff, blksize, 0);

4759                    /*
4760                     * Some other thread has entered the page,
4761                     * so just use it.
4762                     */
4763                    if (pp == NULL)
4764                            goto again;

4766                    /*
4767                     * Now round the request size up to page boundaries.
4768                     * This ensures that the entire page will be
4769                     * initialized to zeroes if EOF is encountered.
4770                     */
4771                    io_len = ptob(btopr(io_len));

4773                    bp = pageio_setup(pp, io_len, vp, B_READ);
4774                    ASSERT(bp != NULL);

4776                    /*
4777                     * pageio_setup should have set b_addr to 0.  This
4778                     * is correct since we want to do I/O on a page
4779                     * boundary.  bp_mapin will use this addr to calculate
4780                     * an offset, and then set b_addr to the kernel virtual
4781                     * address it allocated for us.
4782                     */
4783                    ASSERT(bp->b_un.b_addr == 0);

4785                    bp->b_edev = 0;
4786                    bp->b_dev = 0;
4787                    bp->b_lblkno = lbtodb(io_off);
4788                    bp->b_file = vp;
4789                    bp->b_offset = (offset_t)off;
4790                    bp_mapin(bp);

4792                    /*
4793                     * If doing a write beyond what we believe is EOF,
4794                     * don't bother trying to read the pages from the
4795                     * server, we'll just zero the pages here.  We
4796                     * don't check that the rw flag is S_WRITE here
4797                     * because some implementations may attempt a
4798                     * read access to the buffer before copying data.
4799                     */
4800                    mutex_enter(&rp->r_statelock);
4801                    if (io_off >= rp->r_size && seg == segkmap) {
4802                            mutex_exit(&rp->r_statelock);
4803                            bzero(bp->b_un.b_addr, io_len);
4804                    } else {
4805                            mutex_exit(&rp->r_statelock);
4806                            error = nfs3_bio(bp, NULL, cr);
4807                    }
```



```
4809                            /*
4810                             * Unmap the buffer before freeing it.
4811                             */
4812                            bp_mapout(bp);
4813                            pageio_done(bp);

4815                            savepp = pp;
4816                            do {
4817                                    pp->p_fsdata = C_NOCOMMIT;
4818                            } while ((pp = pp->p_next) != savepp);

4820                            if (error == NFS_EOF) {
4821                                    /*
4822                                     * If doing a write system call just return
4823                                     * zeroed pages, else user tried to get pages
4824                                     * beyond EOF, return error.  We don't check
4825                                     * that the rw flag is S_WRITE here because
4826                                     * some implementations may attempt a read
4827                                     * access to the buffer before copying data.
4828                                     */
4829                                    if (seg == segkmap)
4830                                            error = 0;
4831                                    else
4832                                            error = EFAULT;
4833                            }

4835                            if (!readahead_issued && !error) {
4836                                    mutex_enter(&rp->r_statelock);
4837                                    rp->r_nextr = io_off + io_len;
4838                                    mutex_exit(&rp->r_statelock);
4839                            }
4840                    }
4841            }

4843    out:
4844            if (pl == NULL)
4845                    return (error);

4847            if (error) {
4848                    if (pp != NULL)
4849                            pvn_read_done(pp, B_ERROR);
4850                    return (error);
4851            }

4853            if (pagefound) {
4854                    se_t se = (rw == S_CREATE ? SE_EXCL : SE_SHARED);

4856                    /*
4857                     * Page exists in the cache, acquire the appropriate lock.
4858                     * If this fails, start all over again.
4859                     */
4860                    if ((pp = page_lookup(vp, off, se)) == NULL) {
4861    #ifdef DEBUG
4862                            nfs3_lostpage++;
4863    #endif
4864                            goto reread;
4865                    }
4866                    pl[0] = pp;
4867                    pl[1] = NULL;
4868                    return (0);
4869            }

4871            if (pp != NULL)
4872                    pvn_plist_init(pp, pl, plsz, off, io_len, rw);
```

```
4874         return (error);
4875 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  429784 Thu Jan  8 09:14:34 2015
new/usr/src/uts/common/fs/nfs/nfs4_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
   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 /*
  26  * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
  26  * Copyright 2012 Nexenta Systems, Inc. All rights reserved.
  27  */

  29 /*
  30  *      Copyright 1983,1984,1985,1986,1987,1988,1989 AT&T.
  31  *      All Rights Reserved
  32  */

  34 /*
  35  * Copyright (c) 2013, Joyent, Inc. All rights reserved.
  36  */

  38 #include <sys/param.h>
  39 #include <sys/types.h>
  40 #include <sys/systm.h>
  41 #include <sys/cred.h>
  42 #include <sys/time.h>
  43 #include <sys/vnode.h>
  44 #include <sys/vfs.h>
  45 #include <sys/vfs_opreg.h>
  46 #include <sys/file.h>
  47 #include <sys/filio.h>
  48 #include <sys/uio.h>
  49 #include <sys/buf.h>
  50 #include <sys/mman.h>
  51 #include <sys/pathname.h>
  52 #include <sys/dirent.h>
  53 #include <sys/debug.h>
  54 #include <sys/vmsystm.h>
  55 #include <sys/fcntl.h>
  56 #include <sys/flock.h>
  57 #include <sys/swap.h>
  58 #include <sys/errno.h>
  59 #include <sys/strsubr.h>
  60 #include <sys/sysmacros.h>
```



```
  61 #include <sys/kmem.h>
  62 #include <sys/cmn_err.h>
  63 #include <sys/pathconf.h>
  64 #include <sys/utsname.h>
  65 #include <sys/dnlc.h>
  66 #include <sys/acl.h>
  67 #include <sys/systeminfo.h>
  68 #include <sys/policy.h>
  69 #include <sys/sdt.h>
  70 #include <sys/list.h>
  71 #include <sys/stat.h>
  72 #include <sys/zone.h>

  74 #include <rpc/types.h>
  75 #include <rpc/auth.h>
  76 #include <rpc/clnt.h>

  78 #include <nfs/nfs.h>
  79 #include <nfs/nfs_clnt.h>
  80 #include <nfs/nfs_acl.h>
  81 #include <nfs/lm.h>
  82 #include <nfs/nfs4.h>
  83 #include <nfs/nfs4_kprot.h>
  84 #include <nfs/rnode4.h>
  85 #include <nfs/nfs4_clnt.h>

  87 #include <vm/hat.h>
  88 #include <vm/as.h>
  89 #include <vm/page.h>
  90 #include <vm/pvn.h>
  91 #include <vm/seg.h>
  92 #include <vm/seg_map.h>
  93 #include <vm/seg_kpm.h>
  94 #include <vm/seg_vn.h>

  96 #include <fs/fs_subr.h>

  98 #include <sys/ddi.h>
  99 #include <sys/int_fmtio.h>
 100 #include <sys/fs/autofs.h>

 102 typedef struct {
 103         nfs4_ga_res_t   *di_garp;
 104         cred_t          *di_cred;
 105         hrtime_t        di_time_call;
 106 } dirattr_info_t;
```

_____***unchanged_portion_omitted_***

```
9722 /*
9723  * Return all the pages from [off..off+len) in file
9724  */
9725 /* ARGSUSED */
9726 static int
9727 nfs4_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
9728     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
9729         enum seg_rw rw, cred_t *cr, caller_context_t *ct)
9730 {
9731         rnode4_t *rp;
9732         int error;
9733         mntinfo4_t *mi;

9735         if (nfs_zone() != VTOMI4(vp)->mi_zone)
9736                 return (EIO);
9737         rp = VTOR4(vp);
9738         if (IS_SHADOW(vp, rp))
```

```
9739                vp = RTOV4(rp);

9741        if (vp->v_flag & VNOMAP)
9742                return (ENOSYS);

9744        if (protp != NULL)
9745                *protp = PROT_ALL;

9747        /*
9748         * Now validate that the caches are up to date.
9749         */
9750        if (error = nfs4_validate_caches(vp, cr))
9751                return (error);

9753        mi = VTOMI4(vp);
9754 retry:
9755        mutex_enter(&rp->r_statelock);

9757        /*
9758         * Don't create dirty pages faster than they
9759         * can be cleaned so that the system doesn't
9760         * get imbalanced.  If the async queue is
9761         * maxed out, then wait for it to drain before
9762         * creating more dirty pages.  Also, wait for
9763         * any threads doing pagewalks in the vop_getattr
9764         * entry points so that they don't block for
9765         * long periods.
9766         */
9767        if (rw == S_CREATE) {
9768                while ((mi->mi_max_threads != 0 &&
9769                    rp->r_awcount > 2 * mi->mi_max_threads) ||
9770                    rp->r_gcount > 0)
9771                        cv_wait(&rp->r_cv, &rp->r_statelock);
9772        }

9774        /*
9775         * If we are getting called as a side effect of an nfs_write()
9776         * operation the local file size might not be extended yet.
9777         * In this case we want to be able to return pages of zeroes.
9778         */
9779        if (off + len > rp->r_size + PAGEOFFSET && seg != segkmap) {
9780                NFS4_DEBUG(nfs4_pageio_debug,
9781                    (CE_NOTE, "getpage beyond EOF: off=%lld, "
9782                    "len=%llu, size=%llu, attrsize =%llu", off,
9783                    (u_longlong_t)len, rp->r_size, rp->r_attr.va_size));
9784                mutex_exit(&rp->r_statelock);
9785                return (EFAULT);                /* beyond EOF */
9786        }

9788        mutex_exit(&rp->r_statelock);

9790        if (len <= PAGESIZE) {
9791                error = nfs4_getapage(vp, off, len, protp, pl, plsz,
9792                    seg, addr, rw, cr);
9793                NFS4_DEBUG(nfs4_pageio_debug && error,
9794                    (CE_NOTE, "getpage error %d; off=%lld, "
9795                    "len=%lld", error, off, (u_longlong_t)len));
9796        } else {
9790        error = pvn_getpages(nfs4_getapage, vp, off, len, protp,
9791            pl, plsz, seg, addr, rw, cr);
9792        NFS4_DEBUG(nfs4_pageio_debug && error,
9793            (CE_NOTE, "getpages error %d; off=%lld, len=%lld",
9794            error, off, (u_longlong_t)len));
9800                (CE_NOTE, "getpages error %d; off=%lld, "
9801                "len=%lld", error, off, (u_longlong_t)len));
9802        }
```



```
9796        switch (error) {
9797        case NFS_EOF:
9798                nfs4_purge_caches(vp, NFS4_NOPURGE_DNLC, cr, FALSE);
9799                goto retry;
9800        case ESTALE:
9801                nfs4_purge_stale_fh(error, vp, cr);
9802        }

9804        return (error);
9805 }

9807 /*
9808  * Called from pvn_getpages to get a particular page.
9816  * Called from pvn_getpages or nfs4_getpage to get a particular page.
9809  */
9810 /* ARGSUSED */
9811 static int
9812 nfs4_getapage(vnode_t *vp, u_offset_t off, size_t len, uint_t *protp,
9813     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
9814     enum seg_rw rw, cred_t *cr)
9815 {
9816        rnode4_t *rp;
9817        uint_t bsize;
9818        struct buf *bp;
9819        page_t *pp;
9820        u_offset_t lbn;
9821        u_offset_t io_off;
9822        u_offset_t blkoff;
9823        u_offset_t rablkoff;
9824        size_t io_len;
9825        uint_t blksize;
9826        int error;
9827        int readahead;
9828        int readahead_issued = 0;
9829        int ra_window; /* readahead window */
9830        page_t *pagefound;
9831        page_t *savepp;

9833        if (nfs_zone() != VTOMI4(vp)->mi_zone)
9834                return (EIO);

9836        rp = VTOR4(vp);
9837        ASSERT(!IS_SHADOW(vp, rp));
9838        bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);

9840 reread:
9841        bp = NULL;
9842        pp = NULL;
9843        pagefound = NULL;

9845        if (pl != NULL)
9846                pl[0] = NULL;

9848        error = 0;
9849        lbn = off / bsize;
9850        blkoff = lbn * bsize;

9852        /*
9853         * Queueing up the readahead before doing the synchronous read
9854         * results in a significant increase in read throughput because
9855         * of the increased parallelism between the async threads and
9856         * the process context.
9857         */
9858        if ((off & ((vp->v_vfsp->vfs_bsize) - 1)) == 0 &&
9859            rw != S_CREATE &&
```

```
9860             !(vp->v_flag & VNOCACHE)) {
9861                 mutex_enter(&rp->r_statelock);

9863                 /*
9864                  * Calculate the number of readaheads to do.
9865                  * a) No readaheads at offset = 0.
9866                  * b) Do maximum(nfs4_nra) readaheads when the readahead
9867                  *    window is closed.
9868                  * c) Do readaheads between 1 to (nfs4_nra - 1) depending
9869                  *    upon how far the readahead window is open or close.
9870                  * d) No readaheads if rp->r_nextr is not within the scope
9871                  *    of the readahead window (random i/o).
9872                  */

9874                 if (off == 0)
9875                         readahead = 0;
9876                 else if (blkoff == rp->r_nextr)
9877                         readahead = nfs4_nra;
9878                 else if (rp->r_nextr > blkoff &&
9879                     ((ra_window = (rp->r_nextr - blkoff) / bsize)
9880                     <= (nfs4_nra - 1)))
9881                         readahead = nfs4_nra - ra_window;
9882                 else
9883                         readahead = 0;

9885                 rablkoff = rp->r_nextr;
9886                 while (readahead > 0 && rablkoff + bsize < rp->r_size) {
9887                         mutex_exit(&rp->r_statelock);
9888                         if (nfs4_async_readahead(vp, rablkoff + bsize,
9889                             addr + (rablkoff + bsize - off),
9890                             seg, cr, nfs4_readahead) < 0) {
9891                                 mutex_enter(&rp->r_statelock);
9892                                 break;
9893                         }
9894                         readahead--;
9895                         rablkoff += bsize;
9896                         /*
9897                          * Indicate that we did a readahead so
9898                          * readahead offset is not updated
9899                          * by the synchronous read below.
9900                          */
9901                         readahead_issued = 1;
9902                         mutex_enter(&rp->r_statelock);
9903                         /*
9904                          * set readahead offset to
9905                          * offset of last async readahead
9906                          * request.
9907                          */
9908                         rp->r_nextr = rablkoff;
9909                 }
9910                 mutex_exit(&rp->r_statelock);
9911         }

9913 again:
9914         if ((pagefound = page_exists(vp, off)) == NULL) {
9915                 if (pl == NULL) {
9916                         (void) nfs4_async_readahead(vp, blkoff, addr, seg, cr,
9917                             nfs4_readahead);
9918                 } else if (rw == S_CREATE) {
9919                         /*
9920                          * Block for this page is not allocated, or the offset
9921                          * is beyond the current allocation size, or we're
9922                          * allocating a swap slot and the page was not found,
9923                          * so allocate it and return a zero page.
9924                          */
9925                         if ((pp = page_create_va(vp, off,
```



```
9926                             PAGESIZE, PG_WAIT, seg, addr)) == NULL)
9927                                 cmn_err(CE_PANIC, "nfs4_getapage: page_create");
9928                         io_len = PAGESIZE;
9929                         mutex_enter(&rp->r_statelock);
9930                         rp->r_nextr = off + PAGESIZE;
9931                         mutex_exit(&rp->r_statelock);
9932                 } else {
9933                         /*
9934                          * Need to go to server to get a block
9935                          */
9936                         mutex_enter(&rp->r_statelock);
9937                         if (blkoff < rp->r_size &&
9938                             blkoff + bsize > rp->r_size) {
9939                                 /*
9940                                  * If less than a block left in
9941                                  * file read less than a block.
9942                                  */
9943                                 if (rp->r_size <= off) {
9944                                         /*
9945                                          * Trying to access beyond EOF,
9946                                          * set up to get at least one page.
9947                                          */
9948                                         blksize = off + PAGESIZE - blkoff;
9949                                 } else
9950                                         blksize = rp->r_size - blkoff;
9951                         } else if ((off == 0) ||
9952                             (off != rp->r_nextr && !readahead_issued)) {
9953                                 blksize = PAGESIZE;
9954                                 blkoff = off; /* block = page here */
9955                         } else
9956                                 blksize = bsize;
9957                         mutex_exit(&rp->r_statelock);

9959                         pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
9960                             &io_len, blkoff, blksize, 0);

9962                         /*
9963                          * Some other thread has entered the page,
9964                          * so just use it.
9965                          */
9966                         if (pp == NULL)
9967                                 goto again;

9969                         /*
9970                          * Now round the request size up to page boundaries.
9971                          * This ensures that the entire page will be
9972                          * initialized to zeroes if EOF is encountered.
9973                          */
9974                         io_len = ptob(btopr(io_len));

9976                         bp = pageio_setup(pp, io_len, vp, B_READ);
9977                         ASSERT(bp != NULL);

9979                         /*
9980                          * pageio_setup should have set b_addr to 0.  This
9981                          * is correct since we want to do I/O on a page
9982                          * boundary.  bp_mapin will use this addr to calculate
9983                          * an offset, and then set b_addr to the kernel virtual
9984                          * address it allocated for us.
9985                          */
9986                         ASSERT(bp->b_un.b_addr == 0);

9988                         bp->b_edev = 0;
9989                         bp->b_dev = 0;
9990                         bp->b_lblkno = lbtodb(io_off);
9991                         bp->b_file = vp;
```

```
9992                        bp->b_offset = (offset_t)off;
9993                        bp_mapin(bp);

9995                        /*
9996                         * If doing a write beyond what we believe is EOF,
9997                         * don't bother trying to read the pages from the
9998                         * server, we'll just zero the pages here.  We
9999                         * don't check that the rw flag is S_WRITE here
10000                        * because some implementations may attempt a
10001                        * read access to the buffer before copying data.
10002                        */
10003                       mutex_enter(&rp->r_statelock);
10004                       if (io_off >= rp->r_size && seg == segkmap) {
10005                               mutex_exit(&rp->r_statelock);
10006                               bzero(bp->b_un.b_addr, io_len);
10007                       } else {
10008                               mutex_exit(&rp->r_statelock);
10009                               error = nfs4_bio(bp, NULL, cr, FALSE);
10010                       }

10012                       /*
10013                        * Unmap the buffer before freeing it.
10014                        */
10015                       bp_mapout(bp);
10016                       pageio_done(bp);

10018                       savepp = pp;
10019                       do {
10020                               pp->p_fsdata = C_NOCOMMIT;
10021                       } while ((pp = pp->p_next) != savepp);

10023                       if (error == NFS_EOF) {
10024                               /*
10025                                * If doing a write system call just return
10026                                * zeroed pages, else user tried to get pages
10027                                * beyond EOF, return error.  We don't check
10028                                * that the rw flag is S_WRITE here because
10029                                * some implementations may attempt a read
10030                                * access to the buffer before copying data.
10031                                */
10032                               if (seg == segkmap)
10033                                       error = 0;
10034                               else
10035                                       error = EFAULT;
10036                       }

10038                       if (!readahead_issued && !error) {
10039                               mutex_enter(&rp->r_statelock);
10040                               rp->r_nextr = io_off + io_len;
10041                               mutex_exit(&rp->r_statelock);
10042                       }
10043               }
10044       }

10046 out:
10047       if (pl == NULL)
10048               return (error);

10050       if (error) {
10051               if (pp != NULL)
10052                       pvn_read_done(pp, B_ERROR);
10053               return (error);
10054       }

10056       if (pagefound) {
10057               se_t se = (rw == S_CREATE ? SE_EXCL : SE_SHARED);
```



```
10059                  /*
10060                   * Page exists in the cache, acquire the appropriate lock.
10061                   * If this fails, start all over again.
10062                   */
10063                  if ((pp = page_lookup(vp, off, se)) == NULL) {
10064 #ifdef DEBUG
10065                          nfs4_lostpage++;
10066 #endif
10067                          goto reread;
10068                  }
10069                  pl[0] = pp;
10070                  pl[1] = NULL;
10071                  return (0);
10072          }

10074          if (pp != NULL)
10075                  pvn_plist_init(pp, pl, plsz, off, io_len, rw);

10077          return (error);
10078  }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  130899 Thu Jan  8 09:14:35 2015
new/usr/src/uts/common/fs/nfs/nfs_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1990, 2010, Oracle and/or its affiliates. All rights reserved.
 *
 *      Copyright (c) 1983,1984,1985,1986,1987,1988,1989 AT&T.
 *      All rights reserved.
 */

/*
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#include <sys/param.h>
#include <sys/types.h>
#include <sys/systm.h>
#include <sys/cred.h>
#include <sys/time.h>
#include <sys/vnode.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/file.h>
#include <sys/filio.h>
#include <sys/uio.h>
#include <sys/buf.h>
#include <sys/mman.h>
#include <sys/pathname.h>
#include <sys/dirent.h>
#include <sys/debug.h>
#include <sys/vmsystm.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/swap.h>
#include <sys/errno.h>
#include <sys/strsubr.h>
#include <sys/sysmacros.h>
#include <sys/kmem.h>
#include <sys/cmn_err.h>
#include <sys/pathconf.h>
#include <sys/utsname.h>
#include <sys/dnlc.h>
```



```
#include <sys/acl.h>
#include <sys/atomic.h>
#include <sys/policy.h>
#include <sys/sdt.h>

#include <rpc/types.h>
#include <rpc/auth.h>
#include <rpc/clnt.h>

#include <nfs/nfs.h>
#include <nfs/nfs_clnt.h>
#include <nfs/rnode.h>
#include <nfs/nfs_acl.h>
#include <nfs/lm.h>

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

#include <fs/fs_subr.h>

#include <sys/ddi.h>

static int      nfs_rdwrlbn(vnode_t *, page_t *, u_offset_t, size_t, int,
                        cred_t *);
static int      nfswrite(vnode_t *, caddr_t, uint_t, int, cred_t *);
static int      nfsread(vnode_t *, caddr_t, uint_t, int, size_t *, cred_t *);
static int      nfssetattr(vnode_t *, struct vattr *, int, cred_t *);
static int      nfslookup_dnlc(vnode_t *, char *, vnode_t **, cred_t *);
static int      nfslookup_otw(vnode_t *, char *, vnode_t **, cred_t *, int);
static int      nfsrename(vnode_t *, char *, vnode_t *, char *, cred_t *,
                        caller_context_t *);
static int      nfsreaddir(vnode_t *, rddir_cache *, cred_t *);
static int      nfs_bio(struct buf *, cred_t *);
static int      nfs_getapage(vnode_t *, u_offset_t, size_t, uint_t *,
                        page_t *[], size_t, struct seg *, caddr_t,
                        enum seg_rw, cred_t *);
static void     nfs_readahead(vnode_t *, u_offset_t, caddr_t, struct seg *,
                        cred_t *);
static int      nfs_sync_putapage(vnode_t *, page_t *, u_offset_t, size_t,
                        int, cred_t *);
static int      nfs_sync_pageio(vnode_t *, page_t *, u_offset_t, size_t,
                        int, cred_t *);
static void     nfs_delmap_callback(struct as *, void *, uint_t);

/*
 * Error flags used to pass information about certain special errors
 * which need to be handled specially.
 */
#define NFS_EOF                 -98

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

```
 */

static int      nfs_open(vnode_t **, int, cred_t *, caller_context_t *);
static int      nfs_close(vnode_t *, int, int, offset_t, cred_t *,
                        caller_context_t *);
static int      nfs_read(vnode_t *, struct uio *, int, cred_t *,
                        caller_context_t *);
static int      nfs_write(vnode_t *, struct uio *, int, cred_t *,
                        caller_context_t *);
static int      nfs_ioctl(vnode_t *, int, intptr_t, int, cred_t *, int *,
                        caller_context_t *);
static int      nfs_getattr(vnode_t *, struct vattr *, int, cred_t *,
                        caller_context_t *);
static int      nfs_setattr(vnode_t *, struct vattr *, int, cred_t *,
                        caller_context_t *);
static int      nfs_access(vnode_t *, int, int, cred_t *, caller_context_t *);
static int      nfs_accessx(void *, int, cred_t *);
static int      nfs_readlink(vnode_t *, struct uio *, cred_t *,
                        caller_context_t *);
static int      nfs_fsync(vnode_t *, int, cred_t *, caller_context_t *);
static void     nfs_inactive(vnode_t *, cred_t *, caller_context_t *);
static int      nfs_lookup(vnode_t *, char *, vnode_t **, struct pathname *,
                        int, vnode_t *, cred_t *, caller_context_t *,
                        int *, pathname_t *);
static int      nfs_create(vnode_t *, char *, struct vattr *, enum vcexcl,
                        int, vnode_t **, cred_t *, int, caller_context_t *,
                        vsecattr_t *);
static int      nfs_remove(vnode_t *, char *, cred_t *, caller_context_t *,
                        int);
static int      nfs_link(vnode_t *, vnode_t *, char *, cred_t *,
                        caller_context_t *, int);
static int      nfs_rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
                        caller_context_t *, int);
static int      nfs_mkdir(vnode_t *, char *, struct vattr *, vnode_t **,
                        cred_t *, caller_context_t *, int, vsecattr_t *);
static int      nfs_rmdir(vnode_t *, char *, vnode_t *, cred_t *,
                        caller_context_t *, int);
static int      nfs_symlink(vnode_t *, char *, struct vattr *, char *,
                        cred_t *, caller_context_t *, int);
static int      nfs_readdir(vnode_t *, struct uio *, cred_t *, int *,
                        caller_context_t *, int);
static int      nfs_fid(vnode_t *, fid_t *, caller_context_t *);
static int      nfs_rwlock(vnode_t *, int, caller_context_t *);
static void     nfs_rwunlock(vnode_t *, int, caller_context_t *);
static int      nfs_seek(vnode_t *, offset_t, offset_t *, caller_context_t *);
static int      nfs_getpage(vnode_t *, offset_t, size_t, uint_t *,
                        page_t *[], size_t, struct seg *, caddr_t,
                        enum seg_rw, cred_t *, caller_context_t *);
static int      nfs_putpage(vnode_t *, offset_t, size_t, int, cred_t *,
                        caller_context_t *);
static int      nfs_map(vnode_t *, offset_t, struct as *, caddr_t *, size_t,
                        uchar_t, uchar_t, uint_t, cred_t *, caller_context_t *);
static int      nfs_addmap(vnode_t *, offset_t, struct as *, caddr_t, size_t,
                        uchar_t, uchar_t, uint_t, cred_t *, caller_context_t *);
static int      nfs_frlock(vnode_t *, int, struct flock64 *, int, offset_t,
                        struct flk_callback *, cred_t *, caller_context_t *);
static int      nfs_space(vnode_t *, int, struct flock64 *, int, offset_t,
                        cred_t *, caller_context_t *);
static int      nfs_realvp(vnode_t *, vnode_t **, caller_context_t *);
static int      nfs_delmap(vnode_t *, offset_t, struct as *, caddr_t, size_t,
                        uint_t, uint_t, uint_t, cred_t *, caller_context_t *);
static int      nfs_pathconf(vnode_t *, int, ulong_t *, cred_t *,
                        caller_context_t *);
static int      nfs_pageio(vnode_t *, page_t *, u_offset_t, size_t, int,
                        cred_t *, caller_context_t *);
static int      nfs_setsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
```



```
                        caller_context_t *);
static int      nfs_getsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
                        caller_context_t *);
static int      nfs_shrlock(vnode_t *, int, struct shrlock *, int, cred_t *,
                        caller_context_t *);

struct vnodeops *nfs_vnodeops;

const fs_operation_def_t nfs_vnodeops_template[] = {
        VOPNAME_OPEN,           { .vop_open = nfs_open },
        VOPNAME_CLOSE,          { .vop_close = nfs_close },
        VOPNAME_READ,           { .vop_read = nfs_read },
        VOPNAME_WRITE,          { .vop_write = nfs_write },
        VOPNAME_IOCTL,          { .vop_ioctl = nfs_ioctl },
        VOPNAME_GETATTR,        { .vop_getattr = nfs_getattr },
        VOPNAME_SETATTR,        { .vop_setattr = nfs_setattr },
        VOPNAME_ACCESS,         { .vop_access = nfs_access },
        VOPNAME_LOOKUP,         { .vop_lookup = nfs_lookup },
        VOPNAME_CREATE,         { .vop_create = nfs_create },
        VOPNAME_REMOVE,         { .vop_remove = nfs_remove },
        VOPNAME_LINK,           { .vop_link = nfs_link },
        VOPNAME_RENAME,         { .vop_rename = nfs_rename },
        VOPNAME_MKDIR,          { .vop_mkdir = nfs_mkdir },
        VOPNAME_RMDIR,          { .vop_rmdir = nfs_rmdir },
        VOPNAME_READDIR,        { .vop_readdir = nfs_readdir },
        VOPNAME_SYMLINK,        { .vop_symlink = nfs_symlink },
        VOPNAME_READLINK,       { .vop_readlink = nfs_readlink },
        VOPNAME_FSYNC,          { .vop_fsync = nfs_fsync },
        VOPNAME_INACTIVE,       { .vop_inactive = nfs_inactive },
        VOPNAME_FID,            { .vop_fid = nfs_fid },
        VOPNAME_RWLOCK,         { .vop_rwlock = nfs_rwlock },
        VOPNAME_RWUNLOCK,       { .vop_rwunlock = nfs_rwunlock },
        VOPNAME_SEEK,           { .vop_seek = nfs_seek },
        VOPNAME_FRLOCK,         { .vop_frlock = nfs_frlock },
        VOPNAME_SPACE,          { .vop_space = nfs_space },
        VOPNAME_REALVP,         { .vop_realvp = nfs_realvp },
        VOPNAME_GETPAGE,        { .vop_getpage = nfs_getpage },
        VOPNAME_PUTPAGE,        { .vop_putpage = nfs_putpage },
        VOPNAME_MAP,            { .vop_map = nfs_map },
        VOPNAME_ADDMAP,         { .vop_addmap = nfs_addmap },
        VOPNAME_DELMAP,         { .vop_delmap = nfs_delmap },
        VOPNAME_DUMP,           { .vop_dump = nfs_dump },
        VOPNAME_PATHCONF,       { .vop_pathconf = nfs_pathconf },
        VOPNAME_PAGEIO,         { .vop_pageio = nfs_pageio },
        VOPNAME_SETSECATTR,     { .vop_setsecattr = nfs_setsecattr },
        VOPNAME_GETSECATTR,     { .vop_getsecattr = nfs_getsecattr },
        VOPNAME_SHRLOCK,        { .vop_shrlock = nfs_shrlock },
        VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
        NULL,                   NULL
};

/*
 * XXX:  This is referenced in modstubs.s
 */
struct vnodeops *
nfs_getvnodeops(void)
{
        return (nfs_vnodeops);
}

/* ARGSUSED */
static int
nfs_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
{
        int error;
        struct vattr va;
```

```
260         rnode_t *rp;
261         vnode_t *vp;

263         vp = *vpp;
264         rp = VTOR(vp);
265         if (nfs_zone() != VTOMI(vp)->mi_zone)
266                 return (EIO);
267         mutex_enter(&rp->r_statelock);
268         if (rp->r_cred == NULL) {
269                 crhold(cr);
270                 rp->r_cred = cr;
271         }
272         mutex_exit(&rp->r_statelock);

274         /*
275          * If there is no cached data or if close-to-open
276          * consistency checking is turned off, we can avoid
277          * the over the wire getattr.  Otherwise, if the
278          * file system is mounted readonly, then just verify
279          * the caches are up to date using the normal mechanism.
280          * Else, if the file is not mmap'd, then just mark
281          * the attributes as timed out.  They will be refreshed
282          * and the caches validated prior to being used.
283          * Else, the file system is mounted writeable so
284          * force an over the wire GETATTR in order to ensure
285          * that all cached data is valid.
286          */
287         if (vp->v_count > 1 ||
288             ((vn_has_cached_data(vp) || HAVE_RDDIR_CACHE(rp)) &&
289             !(VTOMI(vp)->mi_flags & MI_NOCTO))) {
290                 if (vn_is_readonly(vp))
291                         error = nfs_validate_caches(vp, cr);
292                 else if (rp->r_mapcnt == 0 && vp->v_count == 1) {
293                         PURGE_ATTRCACHE(vp);
294                         error = 0;
295                 } else {
296                         va.va_mask = AT_ALL;
297                         error = nfs_getattr_otw(vp, &va, cr);
298                 }
299         } else
300                 error = 0;

302         return (error);
303 }

305 /* ARGSUSED */
306 static int
307 nfs_close(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
308         caller_context_t *ct)
309 {
310         rnode_t *rp;
311         int error;
312         struct vattr va;

314         /*
315          * zone_enter(2) prevents processes from changing zones with NFS files
316          * open; if we happen to get here from the wrong zone we can't do
317          * anything over the wire.
318          */
319         if (VTOMI(vp)->mi_zone != nfs_zone()) {
320                 /*
321                  * We could attempt to clean up locks, except we're sure
322                  * that the current process didn't acquire any locks on
323                  * the file: any attempt to lock a file belong to another zone
324                  * will fail, and one can't lock an NFS file and then change
325                  * zones, as that fails too.
```



```
326                  *
327                  * Returning an error here is the sane thing to do.  A
328                  * subsequent call to VN_RELE() which translates to a
329                  * nfs_inactive() will clean up state: if the zone of the
330                  * vnode's origin is still alive and kicking, an async worker
331                  * thread will handle the request (from the correct zone), and
332                  * everything (minus the final nfs_getattr_otw() call) should
333                  * be OK. If the zone is going away nfs_async_inactive() will
334                  * throw away cached pages inline.
335                  */
336                 return (EIO);
337         }

339         /*
340          * If we are using local locking for this filesystem, then
341          * release all of the SYSV style record locks.  Otherwise,
342          * we are doing network locking and we need to release all
343          * of the network locks.  All of the locks held by this
344          * process on this file are released no matter what the
345          * incoming reference count is.
346          */
347         if (VTOMI(vp)->mi_flags & MI_LLOCK) {
348                 cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
349                 cleanshares(vp, ttoproc(curthread)->p_pid);
350         } else
351                 nfs_lockrelease(vp, flag, offset, cr);

353         if (count > 1)
354                 return (0);

356         /*
357          * If the file has been 'unlinked', then purge the
358          * DNLC so that this vnode will get reycled quicker
359          * and the .nfs* file on the server will get removed.
360          */
361         rp = VTOR(vp);
362         if (rp->r_unldvp != NULL)
363                 dnlc_purge_vp(vp);

365         /*
366          * If the file was open for write and there are pages,
367          * then if the file system was mounted using the "no-close-
368          *      to-open" semantics, then start an asynchronous flush
369          *      of the all of the pages in the file.
370          * else the file system was not mounted using the "no-close-
371          *      to-open" semantics, then do a synchronous flush and
372          *      commit of all of the dirty and uncommitted pages.
373          *
374          * The asynchronous flush of the pages in the "nocto" path
375          * mostly just associates a cred pointer with the rnode so
376          * writes which happen later will have a better chance of
377          * working.  It also starts the data being written to the
378          * server, but without unnecessarily delaying the application.
379          */
380         if ((flag & FWRITE) && vn_has_cached_data(vp)) {
381                 if ((VTOMI(vp)->mi_flags & MI_NOCTO)) {
382                         error = nfs_putpage(vp, (offset_t)0, 0, B_ASYNC,
383                             cr, ct);
384                         if (error == EAGAIN)
385                                 error = 0;
386                 } else
387                         error = nfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
388                 if (!error) {
389                         mutex_enter(&rp->r_statelock);
390                         error = rp->r_error;
391                         rp->r_error = 0;
```

```
392                         mutex_exit(&rp->r_statelock);
393                 }
394         } else {
395                 mutex_enter(&rp->r_statelock);
396                 error = rp->r_error;
397                 rp->r_error = 0;
398                 mutex_exit(&rp->r_statelock);
399         }

401         /*
402          * If RWRITEATTR is set, then issue an over the wire GETATTR to
403          * refresh the attribute cache with a set of attributes which
404          * weren't returned from a WRITE.  This will enable the close-
405          * to-open processing to work.
406          */
407         if (rp->r_flags & RWRITEATTR)
408                 (void) nfs_getattr_otw(vp, &va, cr);

410         return (error);
411 }

413 /* ARGSUSED */
414 static int
415 nfs_read(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
416         caller_context_t *ct)
417 {
418         rnode_t *rp;
419         u_offset_t off;
420         offset_t diff;
421         int on;
422         size_t n;
423         caddr_t base;
424         uint_t flags;
425         int error;
426         mntinfo_t *mi;

428         rp = VTOR(vp);
429         mi = VTOMI(vp);

431         if (nfs_zone() != mi->mi_zone)
432                 return (EIO);

434         ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));

436         if (vp->v_type != VREG)
437                 return (EISDIR);

439         if (uiop->uio_resid == 0)
440                 return (0);

442         if (uiop->uio_loffset > MAXOFF32_T)
443                 return (EFBIG);

445         if (uiop->uio_loffset < 0 ||
446             uiop->uio_loffset + uiop->uio_resid > MAXOFF32_T)
447                 return (EINVAL);

449         /*
450          * Bypass VM if caching has been disabled (e.g., locking) or if
451          * using client-side direct I/O and the file is not mmap'd and
452          * there are no cached pages.
453          */
454         if ((vp->v_flag & VNOCACHE) ||
455             (((rp->r_flags & RDIRECTIO) || (mi->mi_flags & MI_DIRECTIO)) &&
456             rp->r_mapcnt == 0 && rp->r_inmap == 0 &&
457             !vn_has_cached_data(vp))) {
```



```
458                 size_t bufsize;
459                 size_t resid = 0;

461                 /*
462                  * Let's try to do read in as large a chunk as we can
463                  * (Filesystem (NFS client) bsize if possible/needed).
464                  * For V3, this is 32K and for V2, this is 8K.
465                  */
466                 bufsize = MIN(uiop->uio_resid, VTOMI(vp)->mi_curread);
467                 base = kmem_alloc(bufsize, KM_SLEEP);
468                 do {
469                         n = MIN(uiop->uio_resid, bufsize);
470                         error = nfsread(vp, base, uiop->uio_offset, n,
471                             &resid, cr);
472                         if (!error) {
473                                 n -= resid;
474                                 error = uiomove(base, n, UIO_READ, uiop);
475                         }
476                 } while (!error && uiop->uio_resid > 0 && n > 0);
477                 kmem_free(base, bufsize);
478                 return (error);
479         }

481         error = 0;

483         do {
484                 off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
485                 on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
486                 n = MIN(MAXBSIZE - on, uiop->uio_resid);

488                 error = nfs_validate_caches(vp, cr);
489                 if (error)
490                         break;

492                 mutex_enter(&rp->r_statelock);
493                 while (rp->r_flags & RINCACHEPURGE) {
494                         if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
495                                 mutex_exit(&rp->r_statelock);
496                                 return (EINTR);
497                         }
498                 }
499                 diff = rp->r_size - uiop->uio_loffset;
500                 mutex_exit(&rp->r_statelock);
501                 if (diff <= 0)
502                         break;
503                 if (diff < n)
504                         n = (size_t)diff;

506                 if (vpm_enable) {
507                         /*
508                          * Copy data.
509                          */
510                         error = vpm_data_copy(vp, off + on, n, uiop,
511                             1, NULL, 0, S_READ);
512                 } else {
513                         base = segmap_getmapflt(segkmap, vp, off + on, n,
514                             1, S_READ);
515                         error = uiomove(base + on, n, UIO_READ, uiop);
516                 }

518                 if (!error) {
519                         /*
520                          * If read a whole block or read to eof,
521                          * won't need this buffer again soon.
522                          */
523                         mutex_enter(&rp->r_statelock);
```

```
524                         if (n + on == MAXBSIZE ||
525                             uiop->uio_loffset == rp->r_size)
526                                 flags = SM_DONTNEED;
527                         else
528                                 flags = 0;
529                         mutex_exit(&rp->r_statelock);
530                         if (vpm_enable) {
531                                 error = vpm_sync_pages(vp, off, n, flags);
532                         } else {
533                                 error = segmap_release(segkmap, base, flags);
534                         }
535                 } else {
536                         if (vpm_enable) {
537                                 (void) vpm_sync_pages(vp, off, n, 0);
538                         } else {
539                                 (void) segmap_release(segkmap, base, 0);
540                         }
541                 }
542         } while (!error && uiop->uio_resid > 0);

544         return (error);
545 }

547 /* ARGSUSED */
548 static int
549 nfs_write(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
550         caller_context_t *ct)
551 {
552         rnode_t *rp;
553         u_offset_t off;
554         caddr_t base;
555         uint_t flags;
556         int remainder;
557         size_t n;
558         int on;
559         int error;
560         int resid;
561         offset_t offset;
562         rlim_t limit;
563         mntinfo_t *mi;

565         rp = VTOR(vp);

567         mi = VTOMI(vp);
568         if (nfs_zone() != mi->mi_zone)
569                 return (EIO);
570         if (vp->v_type != VREG)
571                 return (EISDIR);

573         if (uiop->uio_resid == 0)
574                 return (0);

576         if (ioflag & FAPPEND) {
577                 struct vattr va;

579                 /*
580                  * Must serialize if appending.
581                  */
582                 if (nfs_rw_lock_held(&rp->r_rwlock, RW_READER)) {
583                         nfs_rw_exit(&rp->r_rwlock);
584                         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER,
585                             INTR(vp)))
586                                 return (EINTR);
587                 }

589                 va.va_mask = AT_SIZE;
```



```
590                 error = nfsgetattr(vp, &va, cr);
591                 if (error)
592                         return (error);
593                 uiop->uio_loffset = va.va_size;
594         }

596         if (uiop->uio_loffset > MAXOFF32_T)
597                 return (EFBIG);

599         offset = uiop->uio_loffset + uiop->uio_resid;

601         if (uiop->uio_loffset < 0 || offset > MAXOFF32_T)
602                 return (EINVAL);

604         if (uiop->uio_llimit > (rlim64_t)MAXOFF32_T) {
605                 limit = MAXOFF32_T;
606         } else {
607                 limit = (rlim_t)uiop->uio_llimit;
608         }

610         /*
611          * Check to make sure that the process will not exceed
612          * its limit on file size.  It is okay to write up to
613          * the limit, but not beyond.  Thus, the write which
614          * reaches the limit will be short and the next write
615          * will return an error.
616          */
617         remainder = 0;
618         if (offset > limit) {
619                 remainder = offset - limit;
620                 uiop->uio_resid = limit - uiop->uio_offset;
621                 if (uiop->uio_resid <= 0) {
622                         proc_t *p = ttoproc(curthread);

624                         uiop->uio_resid += remainder;
625                         mutex_enter(&p->p_lock);
626                         (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE],
627                             p->p_rctls, p, RCA_UNSAFE_SIGINFO);
628                         mutex_exit(&p->p_lock);
629                         return (EFBIG);
630                 }
631         }

633         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR(vp)))
634                 return (EINTR);

636         /*
637          * Bypass VM if caching has been disabled (e.g., locking) or if
638          * using client-side direct I/O and the file is not mmap'd and
639          * there are no cached pages.
640          */
641         if ((vp->v_flag & VNOCACHE) ||
642             (((rp->r_flags & RDIRECTIO) || (mi->mi_flags & MI_DIRECTIO)) &&
643             rp->r_mapcnt == 0 && rp->r_inmap == 0 &&
644             !vn_has_cached_data(vp))) {
645                 size_t bufsize;
646                 int count;
647                 uint_t org_offset;

649 nfs_fwrite:
650                 if (rp->r_flags & RSTALE) {
651                         resid = uiop->uio_resid;
652                         offset = uiop->uio_loffset;
653                         error = rp->r_error;
654                         /*
655                          * A close may have cleared r_error, if so,
```

```
656                          * propagate ESTALE error return properly
657                          */
658                         if (error == 0)
659                                 error = ESTALE;
660                         goto bottom;
661                 }
662                 bufsize = MIN(uiop->uio_resid, mi->mi_curwrite);
663                 base = kmem_alloc(bufsize, KM_SLEEP);
664                 do {
665                         resid = uiop->uio_resid;
666                         offset = uiop->uio_loffset;
667                         count = MIN(uiop->uio_resid, bufsize);
668                         org_offset = uiop->uio_offset;
669                         error = uiomove(base, count, UIO_WRITE, uiop);
670                         if (!error) {
671                                 error = nfswrite(vp, base, org_offset,
672                                     count, cr);
673                         }
674                 } while (!error && uiop->uio_resid > 0);
675                 kmem_free(base, bufsize);
676                 goto bottom;
677         }

679         do {
680                 off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
681                 on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
682                 n = MIN(MAXBSIZE - on, uiop->uio_resid);

684                 resid = uiop->uio_resid;
685                 offset = uiop->uio_loffset;

687                 if (rp->r_flags & RSTALE) {
688                         error = rp->r_error;
689                         /*
690                          * A close may have cleared r_error, if so,
691                          * propagate ESTALE error return properly
692                          */
693                         if (error == 0)
694                                 error = ESTALE;
695                         break;
696                 }

698                 /*
699                  * Don't create dirty pages faster than they
700                  * can be cleaned so that the system doesn't
701                  * get imbalanced.  If the async queue is
702                  * maxed out, then wait for it to drain before
703                  * creating more dirty pages.  Also, wait for
704                  * any threads doing pagewalks in the vop_getattr
705                  * entry points so that they don't block for
706                  * long periods.
707                  */
708                 mutex_enter(&rp->r_statelock);
709                 while ((mi->mi_max_threads != 0 &&
710                     rp->r_awcount > 2 * mi->mi_max_threads) ||
711                     rp->r_gcount > 0) {
712                         if (INTR(vp)) {
713                                 klwp_t *lwp = ttolwp(curthread);

715                                 if (lwp != NULL)
716                                         lwp->lwp_nostop++;
717                                 if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
718                                         mutex_exit(&rp->r_statelock);
719                                         if (lwp != NULL)
720                                                 lwp->lwp_nostop--;
721                                         error = EINTR;
```



```
722                                         goto bottom;
723                                 }
724                                 if (lwp != NULL)
725                                         lwp->lwp_nostop--;
726                         } else
727                                 cv_wait(&rp->r_cv, &rp->r_statelock);
728                 }
729                 mutex_exit(&rp->r_statelock);

731                 /*
732                  * Touch the page and fault it in if it is not in core
733                  * before segmap_getmapflt or vpm_data_copy can lock it.
734                  * This is to avoid the deadlock if the buffer is mapped
735                  * to the same file through mmap which we want to write.
736                  */
737                 uio_prefaultpages((long)n, uiop);

739                 if (vpm_enable) {
740                         /*
741                          * It will use kpm mappings, so no need to
742                          * pass an address.
743                          */
744                         error = writerp(rp, NULL, n, uiop, 0);
745                 } else  {
746                         if (segmap_kpm) {
747                                 int pon = uiop->uio_loffset & PAGEOFFSET;
748                                 size_t pn = MIN(PAGESIZE - pon,
749                                     uiop->uio_resid);
750                                 int pagecreate;

752                                 mutex_enter(&rp->r_statelock);
753                                 pagecreate = (pon == 0) && (pn == PAGESIZE ||
754                                     uiop->uio_loffset + pn >= rp->r_size);
755                                 mutex_exit(&rp->r_statelock);

757                                 base = segmap_getmapflt(segkmap, vp, off + on,
758                                     pn, !pagecreate, S_WRITE);

760                                 error = writerp(rp, base + pon, n, uiop,
761                                     pagecreate);

763                         } else {
764                                 base = segmap_getmapflt(segkmap, vp, off + on,
765                                     n, 0, S_READ);
766                                 error = writerp(rp, base + on, n, uiop, 0);
767                         }
768                 }

770                 if (!error) {
771                         if (mi->mi_flags & MI_NOAC)
772                                 flags = SM_WRITE;
773                         else if (n + on == MAXBSIZE || IS_SWAPVP(vp)) {
774                                 /*
775                                  * Have written a whole block.
776                                  * Start an asynchronous write
777                                  * and mark the buffer to
778                                  * indicate that it won't be
779                                  * needed again soon.
780                                  */
781                                 flags = SM_WRITE | SM_ASYNC | SM_DONTNEED;
782                         } else
783                                 flags = 0;
784                         if ((ioflag & (FSYNC|FDSYNC)) ||
785                             (rp->r_flags & ROUTOFSPACE)) {
786                                 flags &= ~SM_ASYNC;
787                                 flags |= SM_WRITE;
```

```
788                         }
789                         if (vpm_enable) {
790                                 error = vpm_sync_pages(vp, off, n, flags);
791                         } else {
792                                 error = segmap_release(segkmap, base, flags);
793                         }
794                 } else {
795                         if (vpm_enable) {
796                                 (void) vpm_sync_pages(vp, off, n, 0);
797                         } else {
798                                 (void) segmap_release(segkmap, base, 0);
799                         }
800                         /*
801                          * In the event that we got an access error while
802                          * faulting in a page for a write-only file just
803                          * force a write.
804                          */
805                         if (error == EACCES)
806                                 goto nfs_fwrite;
807                 }
808         } while (!error && uiop->uio_resid > 0);

810 bottom:
811         if (error) {
812                 uiop->uio_resid = resid + remainder;
813                 uiop->uio_loffset = offset;
814         } else
815                 uiop->uio_resid += remainder;

817         nfs_rw_exit(&rp->r_lkserlock);

819         return (error);
820 }

822 /*
823  * Flags are composed of {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED}
824  */
825 static int
826 nfs_rdwrlbn(vnode_t *vp, page_t *pp, u_offset_t off, size_t len,
827         int flags, cred_t *cr)
828 {
829         struct buf *bp;
830         int error;

832         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);
833         bp = pageio_setup(pp, len, vp, flags);
834         ASSERT(bp != NULL);

836         /*
837          * pageio_setup should have set b_addr to 0.  This
838          * is correct since we want to do I/O on a page
839          * boundary.  bp_mapin will use this addr to calculate
840          * an offset, and then set b_addr to the kernel virtual
841          * address it allocated for us.
842          */
843         ASSERT(bp->b_un.b_addr == 0);

845         bp->b_edev = 0;
846         bp->b_dev = 0;
847         bp->b_lblkno = lbtodb(off);
848         bp->b_file = vp;
849         bp->b_offset = (offset_t)off;
850         bp_mapin(bp);

852         error = nfs_bio(bp, cr);
```



```
854         bp_mapout(bp);
855         pageio_done(bp);

857         return (error);
858 }

860 /*
861  * Write to file.  Writes to remote server in largest size
862  * chunks that the server can handle.  Write is synchronous.
863  */
864 static int
865 nfswrite(vnode_t *vp, caddr_t base, uint_t offset, int count, cred_t *cr)
866 {
867         rnode_t *rp;
868         mntinfo_t *mi;
869         struct nfswriteargs wa;
870         struct nfsattrstat ns;
871         int error;
872         int tsize;
873         int douprintf;

875         douprintf = 1;

877         rp = VTOR(vp);
878         mi = VTOMI(vp);

880         ASSERT(nfs_zone() == mi->mi_zone);

882         wa.wa_args = &wa.wa_args_buf;
883         wa.wa_fhandle = *VTOFH(vp);

885         do {
886                 tsize = MIN(mi->mi_curwrite, count);
887                 wa.wa_data = base;
888                 wa.wa_begoff = offset;
889                 wa.wa_totcount = tsize;
890                 wa.wa_count = tsize;
891                 wa.wa_offset = offset;

893                 if (mi->mi_io_kstats) {
894                         mutex_enter(&mi->mi_lock);
895                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
896                         mutex_exit(&mi->mi_lock);
897                 }
898                 wa.wa_mblk = NULL;
899                 do {
900                         error = rfs2call(mi, RFS_WRITE,
901                             xdr_writeargs, (caddr_t)&wa,
902                             xdr_attrstat, (caddr_t)&ns, cr,
903                             &douprintf, &ns.ns_status, 0, NULL);
904                 } while (error == ENFS_TRYAGAIN);
905                 if (mi->mi_io_kstats) {
906                         mutex_enter(&mi->mi_lock);
907                         kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
908                         mutex_exit(&mi->mi_lock);
909                 }

911                 if (!error) {
912                         error = geterrno(ns.ns_status);
913                         /*
914                          * Can't check for stale fhandle and purge caches
915                          * here because pages are held by nfs_getpage.
916                          * Just mark the attribute cache as timed out
917                          * and set RWRITEATTR to indicate that the file
918                          * was modified with a WRITE operation.
919                          */
```

```
920                         if (!error) {
921                                 count -= tsize;
922                                 base += tsize;
923                                 offset += tsize;
924                                 if (mi->mi_io_kstats) {
925                                         mutex_enter(&mi->mi_lock);
926                                         KSTAT_IO_PTR(mi->mi_io_kstats)->
927                                             writes++;
928                                         KSTAT_IO_PTR(mi->mi_io_kstats)->
929                                             nwritten += tsize;
930                                         mutex_exit(&mi->mi_lock);
931                                 }
932                                 lwp_stat_update(LWP_STAT_OUBLK, 1);
933                                 mutex_enter(&rp->r_statelock);
934                                 PURGE_ATTRCACHE_LOCKED(rp);
935                                 rp->r_flags |= RWRITEATTR;
936                                 mutex_exit(&rp->r_statelock);
937                         }
938                 }
939         } while (!error && count);

941         return (error);
942 }

944 /*
945  * Read from a file.  Reads data in largest chunks our interface can handle.
946  */
947 static int
948 nfsread(vnode_t *vp, caddr_t base, uint_t offset,
949     int count, size_t *residp, cred_t *cr)
950 {
951         mntinfo_t *mi;
952         struct nfsreadargs ra;
953         struct nfsrdresult rr;
954         int tsize;
955         int error;
956         int douprintf;
957         failinfo_t fi;
958         rnode_t *rp;
959         struct vattr va;
960         hrtime_t t;

962         rp = VTOR(vp);
963         mi = VTOMI(vp);

965         ASSERT(nfs_zone() == mi->mi_zone);

967         douprintf = 1;

969         ra.ra_fhandle = *VTOFH(vp);

971         fi.vp = vp;
972         fi.fhp = (caddr_t)&ra.ra_fhandle;
973         fi.copyproc = nfscopyfh;
974         fi.lookupproc = nfslookup;
975         fi.xattrdirproc = acl_getxattrdir2;

977         do {
978                 if (mi->mi_io_kstats) {
979                         mutex_enter(&mi->mi_lock);
980                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
981                         mutex_exit(&mi->mi_lock);
982                 }

984                 do {
985                         tsize = MIN(mi->mi_curread, count);
```



```
986                         rr.rr_data = base;
987                         ra.ra_offset = offset;
988                         ra.ra_totcount = tsize;
989                         ra.ra_count = tsize;
990                         ra.ra_data = base;
991                         t = gethrtime();
992                         error = rfs2call(mi, RFS_READ,
993                             xdr_readargs, (caddr_t)&ra,
994                             xdr_rdresult, (caddr_t)&rr, cr,
995                             &douprintf, &rr.rr_status, 0, &fi);
996                 } while (error == ENFS_TRYAGAIN);

998                 if (mi->mi_io_kstats) {
999                         mutex_enter(&mi->mi_lock);
1000                        kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
1001                        mutex_exit(&mi->mi_lock);
1002                }

1004                if (!error) {
1005                        error = geterrno(rr.rr_status);
1006                        if (!error) {
1007                                count -= rr.rr_count;
1008                                base += rr.rr_count;
1009                                offset += rr.rr_count;
1010                                if (mi->mi_io_kstats) {
1011                                        mutex_enter(&mi->mi_lock);
1012                                        KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
1013                                        KSTAT_IO_PTR(mi->mi_io_kstats)->nread +=
1014                                            rr.rr_count;
1015                                        mutex_exit(&mi->mi_lock);
1016                                }
1017                                lwp_stat_update(LWP_STAT_INBLK, 1);
1018                        }
1019                }
1020        } while (!error && count && rr.rr_count == tsize);

1022        *residp = count;

1024        if (!error) {
1025                /*
1026                 * Since no error occurred, we have the current
1027                 * attributes and we need to do a cache check and then
1028                 * potentially update the cached attributes.  We can't
1029                 * use the normal attribute check and cache mechanisms
1030                 * because they might cause a cache flush which would
1031                 * deadlock.  Instead, we just check the cache to see
1032                 * if the attributes have changed.  If it is, then we
1033                 * just mark the attributes as out of date.  The next
1034                 * time that the attributes are checked, they will be
1035                 * out of date, new attributes will be fetched, and
1036                 * the page cache will be flushed.  If the attributes
1037                 * weren't changed, then we just update the cached
1038                 * attributes with these attributes.
1039                 */
1040                /*
1041                 * If NFS_ACL is supported on the server, then the
1042                 * attributes returned by server may have minimal
1043                 * permissions sometimes denying access to users having
1044                 * proper access.  To get the proper attributes, mark
1045                 * the attributes as expired so that they will be
1046                 * regotten via the NFS_ACL GETATTR2 procedure.
1047                 */
1048                error = nattr_to_vattr(vp, &rr.rr_attr, &va);
1049                mutex_enter(&rp->r_statelock);
1050                if (error || !CACHE_VALID(rp, va.va_mtime, va.va_size) ||
1051                    (mi->mi_flags & MI_ACL)) {
```

```
1052                        mutex_exit(&rp->r_statelock);
1053                        PURGE_ATTRCACHE(vp);
1054                } else {
1055                        if (rp->r_mtime <= t) {
1056                                nfs_attrcache_va(vp, &va);
1057                        }
1058                        mutex_exit(&rp->r_statelock);
1059                }
1060        }

1062        return (error);
1063 }

1065 /* ARGSUSED */
1066 static int
1067 nfs_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp,
1068         caller_context_t *ct)
1069 {

1071        if (nfs_zone() != VTOMI(vp)->mi_zone)
1072                return (EIO);
1073        switch (cmd) {
1074                case _FIODIRECTIO:
1075                        return (nfs_directio(vp, (int)arg, cr));
1076                default:
1077                        return (ENOTTY);
1078        }
1079 }

1081 /* ARGSUSED */
1082 static int
1083 nfs_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1084         caller_context_t *ct)
1085 {
1086        int error;
1087        rnode_t *rp;

1089        if (nfs_zone() != VTOMI(vp)->mi_zone)
1090                return (EIO);
1091        /*
1092         * If it has been specified that the return value will
1093         * just be used as a hint, and we are only being asked
1094         * for size, fsid or rdevid, then return the client's
1095         * notion of these values without checking to make sure
1096         * that the attribute cache is up to date.
1097         * The whole point is to avoid an over the wire GETATTR
1098         * call.
1099         */
1100        rp = VTOR(vp);
1101        if (flags & ATTR_HINT) {
1102                if (vap->va_mask ==
1103                    (vap->va_mask & (AT_SIZE | AT_FSID | AT_RDEV))) {
1104                        mutex_enter(&rp->r_statelock);
1105                        if (vap->va_mask | AT_SIZE)
1106                                vap->va_size = rp->r_size;
1107                        if (vap->va_mask | AT_FSID)
1108                                vap->va_fsid = rp->r_attr.va_fsid;
1109                        if (vap->va_mask | AT_RDEV)
1110                                vap->va_rdev = rp->r_attr.va_rdev;
1111                        mutex_exit(&rp->r_statelock);
1112                        return (0);
1113                }
1114        }

1116        /*
1117         * Only need to flush pages if asking for the mtime
```



```
1118         * and if there any dirty pages or any outstanding
1119         * asynchronous (write) requests for this file.
1120         */
1121        if (vap->va_mask & AT_MTIME) {
1122                if (vn_has_cached_data(vp) &&
1123                    ((rp->r_flags & RDIRTY) || rp->r_awcount > 0)) {
1124                        mutex_enter(&rp->r_statelock);
1125                        rp->r_gcount++;
1126                        mutex_exit(&rp->r_statelock);
1127                        error = nfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
1128                        mutex_enter(&rp->r_statelock);
1129                        if (error && (error == ENOSPC || error == EDQUOT)) {
1130                                if (!rp->r_error)
1131                                        rp->r_error = error;
1132                        }
1133                        if (--rp->r_gcount == 0)
1134                                cv_broadcast(&rp->r_cv);
1135                        mutex_exit(&rp->r_statelock);
1136                }
1137        }

1139        return (nfsgetattr(vp, vap, cr));
1140 }

1142 /*ARGSUSED4*/
1143 static int
1144 nfs_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
1145                 caller_context_t *ct)
1146 {
1147        int error;
1148        uint_t mask;
1149        struct vattr va;

1151        mask = vap->va_mask;

1153        if (mask & AT_NOSET)
1154                return (EINVAL);

1156        if ((mask & AT_SIZE) &&
1157            vap->va_type == VREG &&
1158            vap->va_size > MAXOFF32_T)
1159                return (EFBIG);

1161        if (nfs_zone() != VTOMI(vp)->mi_zone)
1162                return (EIO);

1164        va.va_mask = AT_UID | AT_MODE;

1166        error = nfsgetattr(vp, &va, cr);
1167        if (error)
1168                return (error);

1170        error = secpolicy_vnode_setattr(cr, vp, vap, &va, flags, nfs_accessx,
1171            vp);

1173        if (error)
1174                return (error);

1176        error = nfssetattr(vp, vap, flags, cr);

1178        if (error == 0 && (mask & AT_SIZE) && vap->va_size == 0)
1179                vnevent_truncate(vp, ct);

1181        return (error);
1182 }
```

```
1184 static int
1185 nfssetattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr)
1186 {
1187         int error;
1188         uint_t mask;
1189         struct nfssaargs args;
1190         struct nfsattrstat ns;
1191         int douprintf;
1192         rnode_t *rp;
1193         struct vattr va;
1194         mode_t omode;
1195         mntinfo_t *mi;
1196         vsecattr_t *vsp;
1197         hrtime_t t;

1199         mask = vap->va_mask;

1201         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);

1203         rp = VTOR(vp);

1205         /*
1206          * Only need to flush pages if there are any pages and
1207          * if the file is marked as dirty in some fashion.  The
1208          * file must be flushed so that we can accurately
1209          * determine the size of the file and the cached data
1210          * after the SETATTR returns.  A file is considered to
1211          * be dirty if it is either marked with RDIRTY, has
1212          * outstanding i/o's active, or is mmap'd.  In this
1213          * last case, we can't tell whether there are dirty
1214          * pages, so we flush just to be sure.
1215          */
1216         if (vn_has_cached_data(vp) &&
1217             ((rp->r_flags & RDIRTY) ||
1218             rp->r_count > 0 ||
1219             rp->r_mapcnt > 0)) {
1220                 ASSERT(vp->v_type != VCHR);
1221                 error = nfs_putpage(vp, (offset_t)0, 0, 0, cr, NULL);
1222                 if (error && (error == ENOSPC || error == EDQUOT)) {
1223                         mutex_enter(&rp->r_statelock);
1224                         if (!rp->r_error)
1225                                 rp->r_error = error;
1226                         mutex_exit(&rp->r_statelock);
1227                 }
1228         }

1230         /*
1231          * If the system call was utime(2) or utimes(2) and the
1232          * application did not specify the times, then set the
1233          * mtime nanosecond field to 1 billion.  This will get
1234          * translated from 1 billion nanoseconds to 1 million
1235          * microseconds in the over the wire request.  The
1236          * server will use 1 million in the microsecond field
1237          * to tell whether both the mtime and atime should be
1238          * set to the server's current time.
1239          *
1240          * This is an overload of the protocol and should be
1241          * documented in the NFS Version 2 protocol specification.
1242          */
1243         if ((mask & AT_MTIME) && !(flags & ATTR_UTIME)) {
1244                 vap->va_mtime.tv_nsec = 1000000000;
1245                 if (NFS_TIME_T_OK(vap->va_mtime.tv_sec) &&
1246                     NFS_TIME_T_OK(vap->va_atime.tv_sec)) {
1247                         error = vattr_to_sattr(vap, &args.saa_sa);
1248                 } else {
1249                         /*
```



```
1250                          * Use server times. vap time values will not be used.
1251                          * To ensure no time overflow, make sure vap has
1252                          * valid values, but retain the original values.
1253                          */
1254                         timestruc_t     mtime = vap->va_mtime;
1255                         timestruc_t     atime = vap->va_atime;
1256                         time_t          now;

1258                         now = gethrestime_sec();
1259                         if (NFS_TIME_T_OK(now)) {
1260                                 /* Just in case server does not know of this */
1261                                 vap->va_mtime.tv_sec = now;
1262                                 vap->va_atime.tv_sec = now;
1263                         } else {
1264                                 vap->va_mtime.tv_sec = 0;
1265                                 vap->va_atime.tv_sec = 0;
1266                         }
1267                         error = vattr_to_sattr(vap, &args.saa_sa);
1268                         /* set vap times back on */
1269                         vap->va_mtime = mtime;
1270                         vap->va_atime = atime;
1271                 }
1272         } else {
1273                 /* Either do not set times or use the client specified times */
1274                 error = vattr_to_sattr(vap, &args.saa_sa);
1275         }
1276         if (error) {
1277                 /* req time field(s) overflow - return immediately */
1278                 return (error);
1279         }
1280         args.saa_fh = *VTOFH(vp);

1282         va.va_mask = AT_MODE;
1283         error = nfsgetattr(vp, &va, cr);
1284         if (error)
1285                 return (error);
1286         omode = va.va_mode;

1288         mi = VTOMI(vp);

1290         douprintf = 1;

1292         t = gethrtime();

1294         error = rfs2call(mi, RFS_SETATTR,
1295             xdr_saargs, (caddr_t)&args,
1296             xdr_attrstat, (caddr_t)&ns, cr,
1297             &douprintf, &ns.ns_status, 0, NULL);

1299         /*
1300          * Purge the access cache and ACL cache if changing either the
1301          * owner of the file, the group owner, or the mode.  These may
1302          * change the access permissions of the file, so purge old
1303          * information and start over again.
1304          */
1305         if ((mask & (AT_UID | AT_GID | AT_MODE)) && (mi->mi_flags & MI_ACL)) {
1306                 (void) nfs_access_purge_rp(rp);
1307                 if (rp->r_secattr != NULL) {
1308                         mutex_enter(&rp->r_statelock);
1309                         vsp = rp->r_secattr;
1310                         rp->r_secattr = NULL;
1311                         mutex_exit(&rp->r_statelock);
1312                         if (vsp != NULL)
1313                                 nfs_acl_free(vsp);
1314                 }
1315         }
```

```
1317         if (!error) {
1318                 error = geterrno(ns.ns_status);
1319                 if (!error) {
1320                         /*
1321                          * If changing the size of the file, invalidate
1322                          * any local cached data which is no longer part
1323                          * of the file.  We also possibly invalidate the
1324                          * last page in the file.  We could use
1325                          * pvn_vpzero(), but this would mark the page as
1326                          * modified and require it to be written back to
1327                          * the server for no particularly good reason.
1328                          * This way, if we access it, then we bring it
1329                          * back in.  A read should be cheaper than a
1330                          * write.
1331                          */
1332                         if (mask & AT_SIZE) {
1333                                 nfs_invalidate_pages(vp,
1334                                     (vap->va_size & PAGEMASK), cr);
1335                         }
1336                         (void) nfs_cache_fattr(vp, &ns.ns_attr, &va, t, cr);
1337                         /*
1338                          * If NFS_ACL is supported on the server, then the
1339                          * attributes returned by server may have minimal
1340                          * permissions sometimes denying access to users having
1341                          * proper access.  To get the proper attributes, mark
1342                          * the attributes as expired so that they will be
1343                          * regotten via the NFS_ACL GETATTR2 procedure.
1344                          */
1345                         if (mi->mi_flags & MI_ACL) {
1346                                 PURGE_ATTRCACHE(vp);
1347                         }
1348                         /*
1349                          * This next check attempts to deal with NFS
1350                          * servers which can not handle increasing
1351                          * the size of the file via setattr.  Most
1352                          * of these servers do not return an error,
1353                          * but do not change the size of the file.
1354                          * Hence, this check and then attempt to set
1355                          * the file size by writing 1 byte at the
1356                          * offset of the end of the file that we need.
1357                          */
1358                         if ((mask & AT_SIZE) &&
1359                             ns.ns_attr.na_size < (uint32_t)vap->va_size) {
1360                                 char zb = '\0';

1362                                 error = nfswrite(vp, &zb,
1363                                     vap->va_size - sizeof (zb),
1364                                     sizeof (zb), cr);
1365                         }
1366                         /*
1367                          * Some servers will change the mode to clear the setuid
1368                          * and setgid bits when changing the uid or gid.  The
1369                          * client needs to compensate appropriately.
1370                          */
1371                         if (mask & (AT_UID | AT_GID)) {
1372                                 int terror;

1374                                 va.va_mask = AT_MODE;
1375                                 terror = nfsgetattr(vp, &va, cr);
1376                                 if (!terror &&
1377                                     (((mask & AT_MODE) &&
1378                                     va.va_mode != vap->va_mode) ||
1379                                     (!(mask & AT_MODE) &&
1380                                     va.va_mode != omode))) {
1381                                         va.va_mask = AT_MODE;
```



```
1382                                         if (mask & AT_MODE)
1383                                                 va.va_mode = vap->va_mode;
1384                                         else
1385                                                 va.va_mode = omode;
1386                                         (void) nfssetattr(vp, &va, 0, cr);
1387                                 }
1388                         }
1389                 } else {
1390                         PURGE_ATTRCACHE(vp);
1391                         PURGE_STALE_FH(error, vp, cr);
1392                 }
1393         } else {
1394                 PURGE_ATTRCACHE(vp);
1395         }

1397         return (error);
1398 }

1400 static int
1401 nfs_accessx(void *vp, int mode, cred_t *cr)
1402 {
1403         ASSERT(nfs_zone() == VTOMI((vnode_t *)vp)->mi_zone);
1404         return (nfs_access(vp, mode, 0, cr, NULL));
1405 }

1407 /* ARGSUSED */
1408 static int
1409 nfs_access(vnode_t *vp, int mode, int flags, cred_t *cr, caller_context_t *ct)
1410 {
1411         struct vattr va;
1412         int error;
1413         mntinfo_t *mi;
1414         int shift = 0;

1416         mi = VTOMI(vp);

1418         if (nfs_zone() != mi->mi_zone)
1419                 return (EIO);
1420         if (mi->mi_flags & MI_ACL) {
1421                 error = acl_access2(vp, mode, flags, cr);
1422                 if (mi->mi_flags & MI_ACL)
1423                         return (error);
1424         }

1426         va.va_mask = AT_MODE | AT_UID | AT_GID;
1427         error = nfsgetattr(vp, &va, cr);
1428         if (error)
1429                 return (error);

1431         /*
1432          * Disallow write attempts on read-only
1433          * file systems, unless the file is a
1434          * device node.
1435          */
1436         if ((mode & VWRITE) && vn_is_readonly(vp) && !IS_DEVVP(vp))
1437                 return (EROFS);

1439         /*
1440          * Disallow attempts to access mandatory lock files.
1441          */
1442         if ((mode & (VWRITE | VREAD | VEXEC)) &&
1443             MANDLOCK(vp, va.va_mode))
1444                 return (EACCES);

1446         /*
1447          * Access check is based on only
```

```
1448          * one of owner, group, public.
1449          * If not owner, then check group.
1450          * If not a member of the group,
1451          * then check public access.
1452          */
1453         if (crgetuid(cr) != va.va_uid) {
1454                 shift += 3;
1455                 if (!groupmember(va.va_gid, cr))
1456                         shift += 3;
1457         }

1459         return (secpolicy_vnode_access2(cr, vp, va.va_uid,
1460             va.va_mode << shift, mode));
1461 }

1463 static int nfs_do_symlink_cache = 1;

1465 /* ARGSUSED */
1466 static int
1467 nfs_readlink(vnode_t *vp, struct uio *uiop, cred_t *cr, caller_context_t *ct)
1468 {
1469         int error;
1470         struct nfsrdlnres rl;
1471         rnode_t *rp;
1472         int douprintf;
1473         failinfo_t fi;

1475         /*
1476          * We want to be consistent with UFS semantics so we will return
1477          * EINVAL instead of ENXIO. This violates the XNFS spec and
1478          * the RFC 1094, which are wrong any way. BUGID 1138002.
1479          */
1480         if (vp->v_type != VLNK)
1481                 return (EINVAL);

1483         if (nfs_zone() != VTOMI(vp)->mi_zone)
1484                 return (EIO);

1486         rp = VTOR(vp);
1487         if (nfs_do_symlink_cache && rp->r_symlink.contents != NULL) {
1488                 error = nfs_validate_caches(vp, cr);
1489                 if (error)
1490                         return (error);
1491                 mutex_enter(&rp->r_statelock);
1492                 if (rp->r_symlink.contents != NULL) {
1493                         error = uiomove(rp->r_symlink.contents,
1494                             rp->r_symlink.len, UIO_READ, uiop);
1495                         mutex_exit(&rp->r_statelock);
1496                         return (error);
1497                 }
1498                 mutex_exit(&rp->r_statelock);
1499         }


1502         rl.rl_data = kmem_alloc(NFS_MAXPATHLEN, KM_SLEEP);

1504         fi.vp = vp;
1505         fi.fhp = NULL;          /* no need to update, filehandle not copied */
1506         fi.copyproc = nfscopyfh;
1507         fi.lookupproc = nfslookup;
1508         fi.xattrdirproc = acl_getxattrdir2;

1510         douprintf = 1;

1512         error = rfs2call(VTOMI(vp), RFS_READLINK,
1513             xdr_readlink, (caddr_t)VTOFH(vp),
```



```
1514             xdr_rdlnres, (caddr_t)&rl, cr,
1515             &douprintf, &rl.rl_status, 0, &fi);

1517         if (error) {

1519                 kmem_free((void *)rl.rl_data, NFS_MAXPATHLEN);
1520                 return (error);
1521         }

1523         error = geterrno(rl.rl_status);
1524         if (!error) {
1525                 error = uiomove(rl.rl_data, (int)rl.rl_count, UIO_READ, uiop);
1526                 if (nfs_do_symlink_cache && rp->r_symlink.contents == NULL) {
1527                         mutex_enter(&rp->r_statelock);
1528                         if (rp->r_symlink.contents == NULL) {
1529                                 rp->r_symlink.contents = rl.rl_data;
1530                                 rp->r_symlink.len = (int)rl.rl_count;
1531                                 rp->r_symlink.size = NFS_MAXPATHLEN;
1532                                 mutex_exit(&rp->r_statelock);
1533                         } else {
1534                                 mutex_exit(&rp->r_statelock);

1536                                 kmem_free((void *)rl.rl_data,
1537                                     NFS_MAXPATHLEN);
1538                         }
1539                 } else {

1541                         kmem_free((void *)rl.rl_data, NFS_MAXPATHLEN);
1542                 }
1543         } else {
1544                 PURGE_STALE_FH(error, vp, cr);

1546                 kmem_free((void *)rl.rl_data, NFS_MAXPATHLEN);
1547         }

1549         /*
1550          * Conform to UFS semantics (see comment above)
1551          */
1552         return (error == ENXIO ? EINVAL : error);
1553 }

1555 /*
1556  * Flush local dirty pages to stable storage on the server.
1557  *
1558  * If FNODSYNC is specified, then there is nothing to do because
1559  * metadata changes are not cached on the client before being
1560  * sent to the server.
1561  */
1562 /* ARGSUSED */
1563 static int
1564 nfs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
1565 {
1566         int error;

1568         if ((syncflag & FNODSYNC) || IS_SWAPVP(vp))
1569                 return (0);

1571         if (nfs_zone() != VTOMI(vp)->mi_zone)
1572                 return (EIO);

1574         error = nfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
1575         if (!error)
1576                 error = VTOR(vp)->r_error;
1577         return (error);
1578 }
```

```
1581 /*
1582  * Weirdness: if the file was removed or the target of a rename
1583  * operation while it was open, it got renamed instead.  Here we
1584  * remove the renamed file.
1585  */
1586 /* ARGSUSED */
1587 static void
1588 nfs_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
1589 {
1590         rnode_t *rp;

1592         ASSERT(vp != DNLC_NO_VNODE);

1594         /*
1595          * If this is coming from the wrong zone, we let someone in the right
1596          * zone take care of it asynchronously.  We can get here due to
1597          * VN_RELE() being called from pageout() or fsflush().  This call may
1598          * potentially turn into an expensive no-op if, for instance, v_count
1599          * gets incremented in the meantime, but it's still correct.
1600          */
1601         if (nfs_zone() != VTOMI(vp)->mi_zone) {
1602                 nfs_async_inactive(vp, cr, nfs_inactive);
1603                 return;
1604         }

1606         rp = VTOR(vp);
1607 redo:
1608         if (rp->r_unldvp != NULL) {
1609                 /*
1610                  * Save the vnode pointer for the directory where the
1611                  * unlinked-open file got renamed, then set it to NULL
1612                  * to prevent another thread from getting here before
1613                  * we're done with the remove.  While we have the
1614                  * statelock, make local copies of the pertinent rnode
1615                  * fields.  If we weren't to do this in an atomic way, the
1616                  * the unl* fields could become inconsistent with respect
1617                  * to each other due to a race condition between this
1618                  * code and nfs_remove().  See bug report 1034328.
1619                  */
1620                 mutex_enter(&rp->r_statelock);
1621                 if (rp->r_unldvp != NULL) {
1622                         vnode_t *unldvp;
1623                         char *unlname;
1624                         cred_t *unlcred;
1625                         struct nfsdiropargs da;
1626                         enum nfsstat status;
1627                         int douprintf;
1628                         int error;

1630                         unldvp = rp->r_unldvp;
1631                         rp->r_unldvp = NULL;
1632                         unlname = rp->r_unlname;
1633                         rp->r_unlname = NULL;
1634                         unlcred = rp->r_unlcred;
1635                         rp->r_unlcred = NULL;
1636                         mutex_exit(&rp->r_statelock);

1638                         /*
1639                          * If there are any dirty pages left, then flush
1640                          * them.  This is unfortunate because they just
1641                          * may get thrown away during the remove operation,
1642                          * but we have to do this for correctness.
1643                          */
1644                         if (vn_has_cached_data(vp) &&
1645                             ((rp->r_flags & RDIRTY) || rp->r_count > 0)) {
```



```
1646                                 ASSERT(vp->v_type != VCHR);
1647                                 error = nfs_putpage(vp, (offset_t)0, 0, 0,
1648                                     cr, ct);
1649                                 if (error) {
1650                                         mutex_enter(&rp->r_statelock);
1651                                         if (!rp->r_error)
1652                                                 rp->r_error = error;
1653                                         mutex_exit(&rp->r_statelock);
1654                                 }
1655                         }

1657                         /*
1658                          * Do the remove operation on the renamed file
1659                          */
1660                         setdiropargs(&da, unlname, unldvp);

1662                         douprintf = 1;

1664                         (void) rfs2call(VTOMI(unldvp), RFS_REMOVE,
1665                             xdr_diropargs, (caddr_t)&da,
1666                             xdr_enum, (caddr_t)&status, unlcred,
1667                             &douprintf, &status, 0, NULL);

1669                         if (HAVE_RDDIR_CACHE(VTOR(unldvp)))
1670                                 nfs_purge_rddir_cache(unldvp);
1671                         PURGE_ATTRCACHE(unldvp);

1673                         /*
1674                          * Release stuff held for the remove
1675                          */
1676                         VN_RELE(unldvp);
1677                         kmem_free(unlname, MAXNAMELEN);
1678                         crfree(unlcred);
1679                         goto redo;
1680                 }
1681                 mutex_exit(&rp->r_statelock);
1682         }

1684         rp_addfree(rp, cr);
1685 }

1687 /*
1688  * Remote file system operations having to do with directory manipulation.
1689  */

1691 /* ARGSUSED */
1692 static int
1693 nfs_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
1694         int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
1695         int *direntflags, pathname_t *realpnp)
1696 {
1697         int error;
1698         vnode_t *vp;
1699         vnode_t *avp = NULL;
1700         rnode_t *drp;

1702         if (nfs_zone() != VTOMI(dvp)->mi_zone)
1703                 return (EPERM);

1705         drp = VTOR(dvp);

1707         /*
1708          * Are we looking up extended attributes?  If so, "dvp" is
1709          * the file or directory for which we want attributes, and
1710          * we need a lookup of the hidden attribute directory
1711          * before we lookup the rest of the path.
```

```
1712          */
1713         if (flags & LOOKUP_XATTR) {
1714                 bool_t cflag = ((flags & CREATE_XATTR_DIR) != 0);
1715                 mntinfo_t *mi;

1717                 mi = VTOMI(dvp);
1718                 if (!(mi->mi_flags & MI_EXTATTR))
1719                         return (EINVAL);

1721                 if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR(dvp)))
1722                         return (EINTR);

1724                 (void) nfslookup_dnlc(dvp, XATTR_DIR_NAME, &avp, cr);
1725                 if (avp == NULL)
1726                         error = acl_getxattrdir2(dvp, &avp, cflag, cr, 0);
1727                 else
1728                         error = 0;

1730                 nfs_rw_exit(&drp->r_rwlock);

1732                 if (error) {
1733                         if (mi->mi_flags & MI_EXTATTR)
1734                                 return (error);
1735                         return (EINVAL);
1736                 }
1737                 dvp = avp;
1738                 drp = VTOR(dvp);
1739         }

1741         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR(dvp))) {
1742                 error = EINTR;
1743                 goto out;
1744         }

1746         error = nfslookup(dvp, nm, vpp, pnp, flags, rdir, cr, 0);

1748         nfs_rw_exit(&drp->r_rwlock);

1750         /*
1751          * If vnode is a device, create special vnode.
1752          */
1753         if (!error && IS_DEVVP(*vpp)) {
1754                 vp = *vpp;
1755                 *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
1756                 VN_RELE(vp);
1757         }

1759 out:
1760         if (avp != NULL)
1761                 VN_RELE(avp);

1763         return (error);
1764 }

1766 static int nfs_lookup_neg_cache = 1;

1768 #ifdef DEBUG
1769 static int nfs_lookup_dnlc_hits = 0;
1770 static int nfs_lookup_dnlc_misses = 0;
1771 static int nfs_lookup_dnlc_neg_hits = 0;
1772 static int nfs_lookup_dnlc_disappears = 0;
1773 static int nfs_lookup_dnlc_lookups = 0;
1774 #endif

1776 /* ARGSUSED */
1777 int
```



```
1778 nfslookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
1779         int flags, vnode_t *rdir, cred_t *cr, int rfscall_flags)
1780 {
1781         int error;

1783         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);

1785         /*
1786          * If lookup is for "", just return dvp.  Don't need
1787          * to send it over the wire, look it up in the dnlc,
1788          * or perform any access checks.
1789          */
1790         if (*nm == '\0') {
1791                 VN_HOLD(dvp);
1792                 *vpp = dvp;
1793                 return (0);
1794         }

1796         /*
1797          * Can't do lookups in non-directories.
1798          */
1799         if (dvp->v_type != VDIR)
1800                 return (ENOTDIR);

1802         /*
1803          * If we're called with RFSCALL_SOFT, it's important that
1804          * the only rfscall is one we make directly; if we permit
1805          * an access call because we're looking up "." or validating
1806          * a dnlc hit, we'll deadlock because that rfscall will not
1807          * have the RFSCALL_SOFT set.
1808          */
1809         if (rfscall_flags & RFSCALL_SOFT)
1810                 goto callit;

1812         /*
1813          * If lookup is for ".", just return dvp.  Don't need
1814          * to send it over the wire or look it up in the dnlc,
1815          * just need to check access.
1816          */
1817         if (strcmp(nm, ".") == 0) {
1818                 error = nfs_access(dvp, VEXEC, 0, cr, NULL);
1819                 if (error)
1820                         return (error);
1821                 VN_HOLD(dvp);
1822                 *vpp = dvp;
1823                 return (0);
1824         }

1826         /*
1827          * Lookup this name in the DNLC.  If there was a valid entry,
1828          * then return the results of the lookup.
1829          */
1830         error = nfslookup_dnlc(dvp, nm, vpp, cr);
1831         if (error || *vpp != NULL)
1832                 return (error);

1834 callit:
1835         error = nfslookup_otw(dvp, nm, vpp, cr, rfscall_flags);

1837         return (error);
1838 }

1840 static int
1841 nfslookup_dnlc(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr)
1842 {
1843         int error;
```

```
1844         vnode_t *vp;

1846         ASSERT(*nm != '\0');
1847         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);

1849         /*
1850          * Lookup this name in the DNLC.  If successful, then validate
1851          * the caches and then recheck the DNLC.  The DNLC is rechecked
1852          * just in case this entry got invalidated during the call
1853          * to nfs_validate_caches.
1854          *
1855          * An assumption is being made that it is safe to say that a
1856          * file exists which may not on the server.  Any operations to
1857          * the server will fail with ESTALE.
1858          */
1859 #ifdef DEBUG
1860         nfs_lookup_dnlc_lookups++;
1861 #endif
1862         vp = dnlc_lookup(dvp, nm);
1863         if (vp != NULL) {
1864                 VN_RELE(vp);
1865                 if (vp == DNLC_NO_VNODE && !vn_is_readonly(dvp)) {
1866                         PURGE_ATTRCACHE(dvp);
1867                 }
1868                 error = nfs_validate_caches(dvp, cr);
1869                 if (error)
1870                         return (error);
1871                 vp = dnlc_lookup(dvp, nm);
1872                 if (vp != NULL) {
1873                         error = nfs_access(dvp, VEXEC, 0, cr, NULL);
1874                         if (error) {
1875                                 VN_RELE(vp);
1876                                 return (error);
1877                         }
1878                         if (vp == DNLC_NO_VNODE) {
1879                                 VN_RELE(vp);
1880 #ifdef DEBUG
1881                                 nfs_lookup_dnlc_neg_hits++;
1882 #endif
1883                                 return (ENOENT);
1884                         }
1885                         *vpp = vp;
1886 #ifdef DEBUG
1887                         nfs_lookup_dnlc_hits++;
1888 #endif
1889                         return (0);
1890                 }
1891 #ifdef DEBUG
1892                 nfs_lookup_dnlc_disappears++;
1893 #endif
1894         }
1895 #ifdef DEBUG
1896         else
1897                 nfs_lookup_dnlc_misses++;
1898 #endif

1900         *vpp = NULL;

1902         return (0);
1903 }

1905 static int
1906 nfslookup_otw(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr,
1907         int rfscall_flags)
1908 {
1909         int error;
```



```
1910         struct nfsdiropargs da;
1911         struct nfsdiropres dr;
1912         int douprintf;
1913         failinfo_t fi;
1914         hrtime_t t;

1916         ASSERT(*nm != '\0');
1917         ASSERT(dvp->v_type == VDIR);
1918         ASSERT(nfs_zone() == VTOMI(dvp)->mi_zone);

1920         setdiropargs(&da, nm, dvp);

1922         fi.vp = dvp;
1923         fi.fhp = NULL;          /* no need to update, filehandle not copied */
1924         fi.copyproc = nfscopyfh;
1925         fi.lookupproc = nfslookup;
1926         fi.xattrdirproc = acl_getxattrdir2;

1928         douprintf = 1;

1930         t = gethrtime();

1932         error = rfs2call(VTOMI(dvp), RFS_LOOKUP,
1933             xdr_diropargs, (caddr_t)&da,
1934             xdr_diropres, (caddr_t)&dr, cr,
1935             &douprintf, &dr.dr_status, rfscall_flags, &fi);

1937         if (!error) {
1938                 error = geterrno(dr.dr_status);
1939                 if (!error) {
1940                         *vpp = makenfsnode(&dr.dr_fhandle, &dr.dr_attr,
1941                             dvp->v_vfsp, t, cr, VTOR(dvp)->r_path, nm);
1942                         /*
1943                          * If NFS_ACL is supported on the server, then the
1944                          * attributes returned by server may have minimal
1945                          * permissions sometimes denying access to users having
1946                          * proper access.  To get the proper attributes, mark
1947                          * the attributes as expired so that they will be
1948                          * regotten via the NFS_ACL GETATTR2 procedure.
1949                          */
1950                         if (VTOMI(*vpp)->mi_flags & MI_ACL) {
1951                                 PURGE_ATTRCACHE(*vpp);
1952                         }
1953                         if (!(rfscall_flags & RFSCALL_SOFT))
1954                                 dnlc_update(dvp, nm, *vpp);
1955                 } else {
1956                         PURGE_STALE_FH(error, dvp, cr);
1957                         if (error == ENOENT && nfs_lookup_neg_cache)
1958                                 dnlc_enter(dvp, nm, DNLC_NO_VNODE);
1959                 }
1960         }

1962         return (error);
1963 }

1965 /* ARGSUSED */
1966 static int
1967 nfs_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
1968         int mode, vnode_t **vpp, cred_t *cr, int lfaware, caller_context_t *ct,
1969         vsecattr_t *vsecp)
1970 {
1971         int error;
1972         struct nfscreatargs args;
1973         struct nfsdiropres dr;
1974         int douprintf;
1975         vnode_t *vp;
```

```
1976         rnode_t *rp;
1977         struct vattr vattr;
1978         rnode_t *drp;
1979         vnode_t *tempvp;
1980         hrtime_t t;

1982         drp = VTOR(dvp);

1984         if (nfs_zone() != VTOMI(dvp)->mi_zone)
1985                 return (EPERM);
1986         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
1987                 return (EINTR);

1989         /*
1990          * We make a copy of the attributes because the caller does not
1991          * expect us to change what va points to.
1992          */
1993         vattr = *va;

1995         /*
1996          * If the pathname is "", just use dvp.  Don't need
1997          * to send it over the wire, look it up in the dnlc,
1998          * or perform any access checks.
1999          */
2000         if (*nm == '\0') {
2001                 error = 0;
2002                 VN_HOLD(dvp);
2003                 vp = dvp;
2004         /*
2005          * If the pathname is ".", just use dvp.  Don't need
2006          * to send it over the wire or look it up in the dnlc,
2007          * just need to check access.
2008          */
2009         } else if (strcmp(nm, ".") == 0) {
2010                 error = nfs_access(dvp, VEXEC, 0, cr, ct);
2011                 if (error) {
2012                         nfs_rw_exit(&drp->r_rwlock);
2013                         return (error);
2014                 }
2015                 VN_HOLD(dvp);
2016                 vp = dvp;
2017         /*
2018          * We need to go over the wire, just to be sure whether the
2019          * file exists or not.  Using the DNLC can be dangerous in
2020          * this case when making a decision regarding existence.
2021          */
2022         } else {
2023                 error = nfslookup_otw(dvp, nm, &vp, cr, 0);
2024         }
2025         if (!error) {
2026                 if (exclusive == EXCL)
2027                         error = EEXIST;
2028                 else if (vp->v_type == VDIR && (mode & VWRITE))
2029                         error = EISDIR;
2030                 else {
2031                         /*
2032                          * If vnode is a device, create special vnode.
2033                          */
2034                         if (IS_DEVVP(vp)) {
2035                                 tempvp = vp;
2036                                 vp = specvp(vp, vp->v_rdev, vp->v_type, cr);
2037                                 VN_RELE(tempvp);
2038                         }
2039                         if (!(error = VOP_ACCESS(vp, mode, 0, cr, ct))) {
2040                                 if ((vattr.va_mask & AT_SIZE) &&
2041                                     vp->v_type == VREG) {
```



```
2042                                         vattr.va_mask = AT_SIZE;
2043                                         error = nfssetattr(vp, &vattr, 0, cr);

2045                                         if (!error) {
2046                                                 /*
2047                                                  * Existing file was truncated;
2048                                                  * emit a create event.
2049                                                  */
2050                                                 vnevent_create(vp, ct);
2051                                         }
2052                                 }
2053                         }
2054                 }
2055                 nfs_rw_exit(&drp->r_rwlock);
2056                 if (error) {
2057                         VN_RELE(vp);
2058                 } else {
2059                         *vpp = vp;
2060                 }
2061                 return (error);
2062         }

2064         ASSERT(vattr.va_mask & AT_TYPE);
2065         if (vattr.va_type == VREG) {
2066                 ASSERT(vattr.va_mask & AT_MODE);
2067                 if (MANDMODE(vattr.va_mode)) {
2068                         nfs_rw_exit(&drp->r_rwlock);
2069                         return (EACCES);
2070                 }
2071         }

2073         dnlc_remove(dvp, nm);

2075         setdiropargs(&args.ca_da, nm, dvp);

2077         /*
2078          * Decide what the group-id of the created file should be.
2079          * Set it in attribute list as advisory...then do a setattr
2080          * if the server didn't get it right the first time.
2081          */
2082         error = setdirgid(dvp, &vattr.va_gid, cr);
2083         if (error) {
2084                 nfs_rw_exit(&drp->r_rwlock);
2085                 return (error);
2086         }
2087         vattr.va_mask |= AT_GID;

2089         /*
2090          * This is a completely gross hack to make mknod
2091          * work over the wire until we can wack the protocol
2092          */
2093 #define IFCHR           0020000         /* character special */
2094 #define IFBLK           0060000         /* block special */
2095 #define IFSOCK          0140000         /* socket */

2097         /*
2098          * dev_t is uint_t in 5.x and short in 4.x. Both 4.x
2099          * supports 8 bit majors. 5.x supports 14 bit majors. 5.x supports 18
2100          * bits in the minor number where 4.x supports 8 bits.  If the 5.x
2101          * minor/major numbers <= 8 bits long, compress the device
2102          * number before sending it. Otherwise, the 4.x server will not
2103          * create the device with the correct device number and nothing can be
2104          * done about this.
2105          */
2106         if (vattr.va_type == VCHR || vattr.va_type == VBLK) {
2107                 dev_t d = vattr.va_rdev;
```

```
2108                dev32_t dev32;

2110                if (vattr.va_type == VCHR)
2111                        vattr.va_mode |= IFCHR;
2112                else
2113                        vattr.va_mode |= IFBLK;

2115                (void) cmpldev(&dev32, d);
2116                if (dev32 & ~((SO4_MAXMAJ << L_BITSMINOR32) | SO4_MAXMIN))
2117                        vattr.va_size = (u_offset_t)dev32;
2118                else
2119                        vattr.va_size = (u_offset_t)nfsv2_cmpdev(d);

2121                vattr.va_mask |= AT_MODE|AT_SIZE;
2122        } else if (vattr.va_type == VFIFO) {
2123                vattr.va_mode |= IFCHR;         /* xtra kludge for namedpipe */
2124                vattr.va_size = (u_offset_t)NFS_FIFO_DEV;       /* blech */
2125                vattr.va_mask |= AT_MODE|AT_SIZE;
2126        } else if (vattr.va_type == VSOCK) {
2127                vattr.va_mode |= IFSOCK;
2128                /*
2129                 * To avoid triggering bugs in the servers set AT_SIZE
2130                 * (all other RFS_CREATE calls set this).
2131                 */
2132                vattr.va_size = 0;
2133                vattr.va_mask |= AT_MODE|AT_SIZE;
2134        }

2136        args.ca_sa = &args.ca_sa_buf;
2137        error = vattr_to_sattr(&vattr, args.ca_sa);
2138        if (error) {
2139                /* req time field(s) overflow - return immediately */
2140                nfs_rw_exit(&drp->r_rwlock);
2141                return (error);
2142        }

2144        douprintf = 1;

2146        t = gethrtime();

2148        error = rfs2call(VTOMI(dvp), RFS_CREATE,
2149            xdr_creatargs, (caddr_t)&args,
2150            xdr_diropres, (caddr_t)&dr, cr,
2151            &douprintf, &dr.dr_status, 0, NULL);

2153        PURGE_ATTRCACHE(dvp);   /* mod time changed */

2155        if (!error) {
2156                error = geterrno(dr.dr_status);
2157                if (!error) {
2158                        if (HAVE_RDDIR_CACHE(drp))
2159                                nfs_purge_rddir_cache(dvp);
2160                        vp = makenfsnode(&dr.dr_fhandle, &dr.dr_attr,
2161                            dvp->v_vfsp, t, cr, NULL, NULL);
2162                        /*
2163                         * If NFS_ACL is supported on the server, then the
2164                         * attributes returned by server may have minimal
2165                         * permissions sometimes denying access to users having
2166                         * proper access.  To get the proper attributes, mark
2167                         * the attributes as expired so that they will be
2168                         * regotten via the NFS_ACL GETATTR2 procedure.
2169                         */
2170                        if (VTOMI(vp)->mi_flags & MI_ACL) {
2171                                PURGE_ATTRCACHE(vp);
2172                        }
2173                        dnlc_update(dvp, nm, vp);
```



```
2174                        rp = VTOR(vp);
2175                        if (vattr.va_size == 0) {
2176                                mutex_enter(&rp->r_statelock);
2177                                rp->r_size = 0;
2178                                mutex_exit(&rp->r_statelock);
2179                                if (vn_has_cached_data(vp)) {
2180                                        ASSERT(vp->v_type != VCHR);
2181                                        nfs_invalidate_pages(vp,
2182                                            (u_offset_t)0, cr);
2183                                }
2184                        }

2186                        /*
2187                         * Make sure the gid was set correctly.
2188                         * If not, try to set it (but don't lose
2189                         * any sleep over it).
2190                         */
2191                        if (vattr.va_gid != rp->r_attr.va_gid) {
2192                                vattr.va_mask = AT_GID;
2193                                (void) nfssetattr(vp, &vattr, 0, cr);
2194                        }

2196                        /*
2197                         * If vnode is a device create special vnode
2198                         */
2199                        if (IS_DEVVP(vp)) {
2200                                *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
2201                                VN_RELE(vp);
2202                        } else
2203                                *vpp = vp;
2204                } else {
2205                        PURGE_STALE_FH(error, dvp, cr);
2206                }
2207        }

2209        nfs_rw_exit(&drp->r_rwlock);

2211        return (error);
2212 }

2214 /*
2215  * Weirdness: if the vnode to be removed is open
2216  * we rename it instead of removing it and nfs_inactive
2217  * will remove the new name.
2218  */
2219 /* ARGSUSED */
2220 static int
2221 nfs_remove(vnode_t *dvp, char *nm, cred_t *cr, caller_context_t *ct, int flags)
2222 {
2223        int error;
2224        struct nfsdiropargs da;
2225        enum nfsstat status;
2226        vnode_t *vp;
2227        char *tmpname;
2228        int douprintf;
2229        rnode_t *rp;
2230        rnode_t *drp;

2232        if (nfs_zone() != VTOMI(dvp)->mi_zone)
2233                return (EPERM);
2234        drp = VTOR(dvp);
2235        if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
2236                return (EINTR);

2238        error = nfslookup(dvp, nm, &vp, NULL, 0, NULL, cr, 0);
2239        if (error) {
```

```
2240                nfs_rw_exit(&drp->r_rwlock);
2241                return (error);
2242        }

2244        if (vp->v_type == VDIR && secpolicy_fs_linkdir(cr, dvp->v_vfsp)) {
2245                VN_RELE(vp);
2246                nfs_rw_exit(&drp->r_rwlock);
2247                return (EPERM);
2248        }

2250        /*
2251         * First just remove the entry from the name cache, as it
2252         * is most likely the only entry for this vp.
2253         */
2254        dnlc_remove(dvp, nm);

2256        /*
2257         * If the file has a v_count > 1 then there may be more than one
2258         * entry in the name cache due multiple links or an open file,
2259         * but we don't have the real reference count so flush all
2260         * possible entries.
2261         */
2262        if (vp->v_count > 1)
2263                dnlc_purge_vp(vp);

2265        /*
2266         * Now we have the real reference count on the vnode
2267         */
2268        rp = VTOR(vp);
2269        mutex_enter(&rp->r_statelock);
2270        if (vp->v_count > 1 &&
2271            (rp->r_unldvp == NULL || strcmp(nm, rp->r_unlname) == 0)) {
2272                mutex_exit(&rp->r_statelock);
2273                tmpname = newname();
2274                error = nfsrename(dvp, nm, dvp, tmpname, cr, ct);
2275                if (error)
2276                        kmem_free(tmpname, MAXNAMELEN);
2277                else {
2278                        mutex_enter(&rp->r_statelock);
2279                        if (rp->r_unldvp == NULL) {
2280                                VN_HOLD(dvp);
2281                                rp->r_unldvp = dvp;
2282                                if (rp->r_unlcred != NULL)
2283                                        crfree(rp->r_unlcred);
2284                                crhold(cr);
2285                                rp->r_unlcred = cr;
2286                                rp->r_unlname = tmpname;
2287                        } else {
2288                                kmem_free(rp->r_unlname, MAXNAMELEN);
2289                                rp->r_unlname = tmpname;
2290                        }
2291                        mutex_exit(&rp->r_statelock);
2292                }
2293        } else {
2294                mutex_exit(&rp->r_statelock);
2295                /*
2296                 * We need to flush any dirty pages which happen to
2297                 * be hanging around before removing the file.  This
2298                 * shouldn't happen very often and mostly on file
2299                 * systems mounted "nocto".
2300                 */
2301                if (vn_has_cached_data(vp) &&
2302                    ((rp->r_flags & RDIRTY) || rp->r_count > 0)) {
2303                        error = nfs_putpage(vp, (offset_t)0, 0, 0, cr, ct);
2304                        if (error && (error == ENOSPC || error == EDQUOT)) {
2305                                mutex_enter(&rp->r_statelock);
```



```
2306                                if (!rp->r_error)
2307                                        rp->r_error = error;
2308                                mutex_exit(&rp->r_statelock);
2309                        }
2310                }

2312                setdiropargs(&da, nm, dvp);

2314                douprintf = 1;

2316                error = rfs2call(VTOMI(dvp), RFS_REMOVE,
2317                    xdr_diropargs, (caddr_t)&da,
2318                    xdr_enum, (caddr_t)&status, cr,
2319                    &douprintf, &status, 0, NULL);

2321                /*
2322                 * The xattr dir may be gone after last attr is removed,
2323                 * so flush it from dnlc.
2324                 */
2325                if (dvp->v_flag & V_XATTRDIR)
2326                        dnlc_purge_vp(dvp);

2328                PURGE_ATTRCACHE(dvp);   /* mod time changed */
2329                PURGE_ATTRCACHE(vp);    /* link count changed */

2331                if (!error) {
2332                        error = geterrno(status);
2333                        if (!error) {
2334                                if (HAVE_RDDIR_CACHE(drp))
2335                                        nfs_purge_rddir_cache(dvp);
2336                        } else {
2337                                PURGE_STALE_FH(error, dvp, cr);
2338                        }
2339                }
2340        }

2342        if (error == 0) {
2343                vnevent_remove(vp, dvp, nm, ct);
2344        }
2345        VN_RELE(vp);

2347        nfs_rw_exit(&drp->r_rwlock);

2349        return (error);
2350 }

2352 /* ARGSUSED */
2353 static int
2354 nfs_link(vnode_t *tdvp, vnode_t *svp, char *tnm, cred_t *cr,
2355         caller_context_t *ct, int flags)
2356 {
2357         int error;
2358         struct nfslinkargs args;
2359         enum nfsstat status;
2360         vnode_t *realvp;
2361         int douprintf;
2362         rnode_t *tdrp;

2364         if (nfs_zone() != VTOMI(tdvp)->mi_zone)
2365                 return (EPERM);
2366         if (VOP_REALVP(svp, &realvp, ct) == 0)
2367                 svp = realvp;

2369         args.la_from = VTOFH(svp);
2370         setdiropargs(&args.la_to, tnm, tdvp);
```

```
2372         tdrp = VTOR(tdvp);
2373         if (nfs_rw_enter_sig(&tdrp->r_rwlock, RW_WRITER, INTR(tdvp)))
2374                 return (EINTR);

2376         dnlc_remove(tdvp, tnm);

2378         douprintf = 1;

2380         error = rfs2call(VTOMI(svp), RFS_LINK,
2381             xdr_linkargs, (caddr_t)&args,
2382             xdr_enum, (caddr_t)&status, cr,
2383             &douprintf, &status, 0, NULL);

2385         PURGE_ATTRCACHE(tdvp);  /* mod time changed */
2386         PURGE_ATTRCACHE(svp);   /* link count changed */

2388         if (!error) {
2389                 error = geterrno(status);
2390                 if (!error) {
2391                         if (HAVE_RDDIR_CACHE(tdrp))
2392                                 nfs_purge_rddir_cache(tdvp);
2393                 }
2394         }

2396         nfs_rw_exit(&tdrp->r_rwlock);

2398         if (!error) {
2399                 /*
2400                  * Notify the source file of this link operation.
2401                  */
2402                 vnevent_link(svp, ct);
2403         }
2404         return (error);
2405 }

2407 /* ARGSUSED */
2408 static int
2409 nfs_rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
2410         caller_context_t *ct, int flags)
2411 {
2412         vnode_t *realvp;

2414         if (nfs_zone() != VTOMI(odvp)->mi_zone)
2415                 return (EPERM);
2416         if (VOP_REALVP(ndvp, &realvp, ct) == 0)
2417                 ndvp = realvp;

2419         return (nfsrename(odvp, onm, ndvp, nnm, cr, ct));
2420 }

2422 /*
2423  * nfsrename does the real work of renaming in NFS Version 2.
2424  */
2425 static int
2426 nfsrename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
2427     caller_context_t *ct)
2428 {
2429         int error;
2430         enum nfsstat status;
2431         struct nfsrnmargs args;
2432         int douprintf;
2433         vnode_t *nvp = NULL;
2434         vnode_t *ovp = NULL;
2435         char *tmpname;
2436         rnode_t *rp;
2437         rnode_t *odrp;
```



```
2438         rnode_t *ndrp;

2440         ASSERT(nfs_zone() == VTOMI(odvp)->mi_zone);
2441         if (strcmp(onm, ".") == 0 || strcmp(onm, "..") == 0 ||
2442             strcmp(nnm, ".") == 0 || strcmp(nnm, "..") == 0)
2443                 return (EINVAL);

2445         odrp = VTOR(odvp);
2446         ndrp = VTOR(ndvp);
2447         if ((intptr_t)odrp < (intptr_t)ndrp) {
2448                 if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR(odvp)))
2449                         return (EINTR);
2450                 if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR(ndvp))) {
2451                         nfs_rw_exit(&odrp->r_rwlock);
2452                         return (EINTR);
2453                 }
2454         } else {
2455                 if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR(ndvp)))
2456                         return (EINTR);
2457                 if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR(odvp))) {
2458                         nfs_rw_exit(&ndrp->r_rwlock);
2459                         return (EINTR);
2460                 }
2461         }

2463         /*
2464          * Lookup the target file.  If it exists, it needs to be
2465          * checked to see whether it is a mount point and whether
2466          * it is active (open).
2467          */
2468         error = nfslookup(ndvp, nnm, &nvp, NULL, 0, NULL, cr, 0);
2469         if (!error) {
2470                 /*
2471                  * If this file has been mounted on, then just
2472                  * return busy because renaming to it would remove
2473                  * the mounted file system from the name space.
2474                  */
2475                 if (vn_mountedvfs(nvp) != NULL) {
2476                         VN_RELE(nvp);
2477                         nfs_rw_exit(&odrp->r_rwlock);
2478                         nfs_rw_exit(&ndrp->r_rwlock);
2479                         return (EBUSY);
2480                 }

2482                 /*
2483                  * Purge the name cache of all references to this vnode
2484                  * so that we can check the reference count to infer
2485                  * whether it is active or not.
2486                  */
2487                 /*
2488                  * First just remove the entry from the name cache, as it
2489                  * is most likely the only entry for this vp.
2490                  */
2491                 dnlc_remove(ndvp, nnm);
2492                 /*
2493                  * If the file has a v_count > 1 then there may be more
2494                  * than one entry in the name cache due multiple links
2495                  * or an open file, but we don't have the real reference
2496                  * count so flush all possible entries.
2497                  */
2498                 if (nvp->v_count > 1)
2499                         dnlc_purge_vp(nvp);

2501                 /*
2502                  * If the vnode is active and is not a directory,
2503                  * arrange to rename it to a
```

```
2504                 * temporary file so that it will continue to be
2505                 * accessible.  This implements the "unlink-open-file"
2506                 * semantics for the target of a rename operation.
2507                 * Before doing this though, make sure that the
2508                 * source and target files are not already the same.
2509                 */
2510                if (nvp->v_count > 1 && nvp->v_type != VDIR) {
2511                        /*
2512                         * Lookup the source name.
2513                         */
2514                        error = nfslookup(odvp, onm, &ovp, NULL, 0, NULL,
2515                            cr, 0);

2517                        /*
2518                         * The source name *should* already exist.
2519                         */
2520                        if (error) {
2521                                VN_RELE(nvp);
2522                                nfs_rw_exit(&odrp->r_rwlock);
2523                                nfs_rw_exit(&ndrp->r_rwlock);
2524                                return (error);
2525                        }

2527                        /*
2528                         * Compare the two vnodes.  If they are the same,
2529                         * just release all held vnodes and return success.
2530                         */
2531                        if (ovp == nvp) {
2532                                VN_RELE(ovp);
2533                                VN_RELE(nvp);
2534                                nfs_rw_exit(&odrp->r_rwlock);
2535                                nfs_rw_exit(&ndrp->r_rwlock);
2536                                return (0);
2537                        }

2539                        /*
2540                         * Can't mix and match directories and non-
2541                         * directories in rename operations.  We already
2542                         * know that the target is not a directory.  If
2543                         * the source is a directory, return an error.
2544                         */
2545                        if (ovp->v_type == VDIR) {
2546                                VN_RELE(ovp);
2547                                VN_RELE(nvp);
2548                                nfs_rw_exit(&odrp->r_rwlock);
2549                                nfs_rw_exit(&ndrp->r_rwlock);
2550                                return (ENOTDIR);
2551                        }

2553                        /*
2554                         * The target file exists, is not the same as
2555                         * the source file, and is active.  Link it
2556                         * to a temporary filename to avoid having
2557                         * the server removing the file completely.
2558                         */
2559                        tmpname = newname();
2560                        error = nfs_link(ndvp, nvp, tmpname, cr, NULL, 0);
2561                        if (error == EOPNOTSUPP) {
2562                                error = nfs_rename(ndvp, nnm, ndvp, tmpname,
2563                                    cr, NULL, 0);
2564                        }
2565                        if (error) {
2566                                kmem_free(tmpname, MAXNAMELEN);
2567                                VN_RELE(ovp);
2568                                VN_RELE(nvp);
2569                                nfs_rw_exit(&odrp->r_rwlock);
```



```
2570                                nfs_rw_exit(&ndrp->r_rwlock);
2571                                return (error);
2572                        }
2573                        rp = VTOR(nvp);
2574                        mutex_enter(&rp->r_statelock);
2575                        if (rp->r_unldvp == NULL) {
2576                                VN_HOLD(ndvp);
2577                                rp->r_unldvp = ndvp;
2578                                if (rp->r_unlcred != NULL)
2579                                        crfree(rp->r_unlcred);
2580                                crhold(cr);
2581                                rp->r_unlcred = cr;
2582                                rp->r_unlname = tmpname;
2583                        } else {
2584                                kmem_free(rp->r_unlname, MAXNAMELEN);
2585                                rp->r_unlname = tmpname;
2586                        }
2587                        mutex_exit(&rp->r_statelock);
2588                }
2589        }

2591        if (ovp == NULL) {
2592                /*
2593                 * When renaming directories to be a subdirectory of a
2594                 * different parent, the dnlc entry for ".." will no
2595                 * longer be valid, so it must be removed.
2596                 *
2597                 * We do a lookup here to determine whether we are renaming
2598                 * a directory and we need to check if we are renaming
2599                 * an unlinked file.  This might have already been done
2600                 * in previous code, so we check ovp == NULL to avoid
2601                 * doing it twice.
2602                 */

2604                error = nfslookup(odvp, onm, &ovp, NULL, 0, NULL, cr, 0);

2606                /*
2607                 * The source name *should* already exist.
2608                 */
2609                if (error) {
2610                        nfs_rw_exit(&odrp->r_rwlock);
2611                        nfs_rw_exit(&ndrp->r_rwlock);
2612                        if (nvp) {
2613                                VN_RELE(nvp);
2614                        }
2615                        return (error);
2616                }
2617                ASSERT(ovp != NULL);
2618        }

2620        dnlc_remove(odvp, onm);
2621        dnlc_remove(ndvp, nnm);

2623        setdiropargs(&args.rna_from, onm, odvp);
2624        setdiropargs(&args.rna_to, nnm, ndvp);

2626        douprintf = 1;

2628        error = rfs2call(VTOMI(odvp), RFS_RENAME,
2629            xdr_rnmargs, (caddr_t)&args,
2630            xdr_enum, (caddr_t)&status, cr,
2631            &douprintf, &status, 0, NULL);

2633        PURGE_ATTRCACHE(odvp);  /* mod time changed */
2634        PURGE_ATTRCACHE(ndvp);  /* mod time changed */
```

```
2636         if (!error) {
2637                 error = geterrno(status);
2638                 if (!error) {
2639                         if (HAVE_RDDIR_CACHE(odrp))
2640                                 nfs_purge_rddir_cache(odvp);
2641                         if (HAVE_RDDIR_CACHE(ndrp))
2642                                 nfs_purge_rddir_cache(ndvp);
2643                         /*
2644                          * when renaming directories to be a subdirectory of a
2645                          * different parent, the dnlc entry for ".." will no
2646                          * longer be valid, so it must be removed
2647                          */
2648                         rp = VTOR(ovp);
2649                         if (ndvp != odvp) {
2650                                 if (ovp->v_type == VDIR) {
2651                                         dnlc_remove(ovp, "..");
2652                                         if (HAVE_RDDIR_CACHE(rp))
2653                                                 nfs_purge_rddir_cache(ovp);
2654                                 }
2655                         }
2657                         /*
2658                          * If we are renaming the unlinked file, update the
2659                          * r_unldvp and r_unlname as needed.
2660                          */
2661                         mutex_enter(&rp->r_statelock);
2662                         if (rp->r_unldvp != NULL) {
2663                                 if (strcmp(rp->r_unlname, onm) == 0) {
2664                                         (void) strncpy(rp->r_unlname,
2665                                             nnm, MAXNAMELEN);
2666                                         rp->r_unlname[MAXNAMELEN - 1] = '\0';
2668                                         if (ndvp != rp->r_unldvp) {
2669                                                 VN_RELE(rp->r_unldvp);
2670                                                 rp->r_unldvp = ndvp;
2671                                                 VN_HOLD(ndvp);
2672                                         }
2673                                 }
2674                         }
2675                         mutex_exit(&rp->r_statelock);
2676                 } else {
2677                         /*
2678                          * System V defines rename to return EEXIST, not
2679                          * ENOTEMPTY if the target directory is not empty.
2680                          * Over the wire, the error is NFSERR_ENOTEMPTY
2681                          * which geterrno maps to ENOTEMPTY.
2682                          */
2683                         if (error == ENOTEMPTY)
2684                                 error = EEXIST;
2685                 }
2686         }
2688         if (error == 0) {
2689                 if (nvp)
2690                         vnevent_rename_dest(nvp, ndvp, nnm, ct);
2692                 if (odvp != ndvp)
2693                         vnevent_rename_dest_dir(ndvp, ct);
2695                 ASSERT(ovp != NULL);
2696                 vnevent_rename_src(ovp, odvp, onm, ct);
2697         }
2699         if (nvp) {
2700                 VN_RELE(nvp);
2701         }
```



```
2702         VN_RELE(ovp);
2704         nfs_rw_exit(&odrp->r_rwlock);
2705         nfs_rw_exit(&ndrp->r_rwlock);
2707         return (error);
2708 }
2710 /* ARGSUSED */
2711 static int
2712 nfs_mkdir(vnode_t *dvp, char *nm, struct vattr *va, vnode_t **vpp, cred_t *cr,
2713         caller_context_t *ct, int flags, vsecattr_t *vsecp)
2714 {
2715         int error;
2716         struct nfscreatargs args;
2717         struct nfsdiropres dr;
2718         int douprintf;
2719         rnode_t *drp;
2720         hrtime_t t;
2722         if (nfs_zone() != VTOMI(dvp)->mi_zone)
2723                 return (EPERM);
2725         setdiropargs(&args.ca_da, nm, dvp);
2727         /*
2728          * Decide what the group-id and set-gid bit of the created directory
2729          * should be.  May have to do a setattr to get the gid right.
2730          */
2731         error = setdirgid(dvp, &va->va_gid, cr);
2732         if (error)
2733                 return (error);
2734         error = setdirmode(dvp, &va->va_mode, cr);
2735         if (error)
2736                 return (error);
2737         va->va_mask |= AT_MODE|AT_GID;
2739         args.ca_sa = &args.ca_sa_buf;
2740         error = vattr_to_sattr(va, args.ca_sa);
2741         if (error) {
2742                 /* req time field(s) overflow - return immediately */
2743                 return (error);
2744         }
2746         drp = VTOR(dvp);
2747         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
2748                 return (EINTR);
2750         dnlc_remove(dvp, nm);
2752         douprintf = 1;
2754         t = gethrtime();
2756         error = rfs2call(VTOMI(dvp), RFS_MKDIR,
2757             xdr_creatargs, (caddr_t)&args,
2758             xdr_diropres, (caddr_t)&dr, cr,
2759             &douprintf, &dr.dr_status, 0, NULL);
2761         PURGE_ATTRCACHE(dvp);   /* mod time changed */
2763         if (!error) {
2764                 error = geterrno(dr.dr_status);
2765                 if (!error) {
2766                         if (HAVE_RDDIR_CACHE(drp))
2767                                 nfs_purge_rddir_cache(dvp);
```

```
2768                        /*
2769                         * The attributes returned by RFS_MKDIR can not
2770                         * be depended upon, so mark the attribute cache
2771                         * as purged.  A subsequent GETATTR will get the
2772                         * correct attributes from the server.
2773                         */
2774                        *vpp = makenfsnode(&dr.dr_fhandle, &dr.dr_attr,
2775                            dvp->v_vfsp, t, cr, NULL, NULL);
2776                        PURGE_ATTRCACHE(*vpp);
2777                        dnlc_update(dvp, nm, *vpp);

2779                        /*
2780                         * Make sure the gid was set correctly.
2781                         * If not, try to set it (but don't lose
2782                         * any sleep over it).
2783                         */
2784                        if (va->va_gid != VTOR(*vpp)->r_attr.va_gid) {
2785                                va->va_mask = AT_GID;
2786                                (void) nfssetattr(*vpp, va, 0, cr);
2787                        }
2788                } else {
2789                        PURGE_STALE_FH(error, dvp, cr);
2790                }
2791        }

2793        nfs_rw_exit(&drp->r_rwlock);

2795        return (error);
2796 }

2798 /* ARGSUSED */
2799 static int
2800 nfs_rmdir(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
2801        caller_context_t *ct, int flags)
2802 {
2803        int error;
2804        enum nfsstat status;
2805        struct nfsdiropargs da;
2806        vnode_t *vp;
2807        int douprintf;
2808        rnode_t *drp;

2810        if (nfs_zone() != VTOMI(dvp)->mi_zone)
2811                return (EPERM);
2812        drp = VTOR(dvp);
2813        if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
2814                return (EINTR);

2816        /*
2817         * Attempt to prevent a rmdir(".") from succeeding.
2818         */
2819        error = nfslookup(dvp, nm, &vp, NULL, 0, NULL, cr, 0);
2820        if (error) {
2821                nfs_rw_exit(&drp->r_rwlock);
2822                return (error);
2823        }

2825        if (vp == cdir) {
2826                VN_RELE(vp);
2827                nfs_rw_exit(&drp->r_rwlock);
2828                return (EINVAL);
2829        }

2831        setdiropargs(&da, nm, dvp);

2833        /*
```



```
2834         * First just remove the entry from the name cache, as it
2835         * is most likely an entry for this vp.
2836         */
2837        dnlc_remove(dvp, nm);

2839        /*
2840         * If there vnode reference count is greater than one, then
2841         * there may be additional references in the DNLC which will
2842         * need to be purged.  First, trying removing the entry for
2843         * the parent directory and see if that removes the additional
2844         * reference(s).  If that doesn't do it, then use dnlc_purge_vp
2845         * to completely remove any references to the directory which
2846         * might still exist in the DNLC.
2847         */
2848        if (vp->v_count > 1) {
2849                dnlc_remove(vp, "..");
2850                if (vp->v_count > 1)
2851                        dnlc_purge_vp(vp);
2852        }

2854        douprintf = 1;

2856        error = rfs2call(VTOMI(dvp), RFS_RMDIR,
2857            xdr_diropargs, (caddr_t)&da,
2858            xdr_enum, (caddr_t)&status, cr,
2859            &douprintf, &status, 0, NULL);

2861        PURGE_ATTRCACHE(dvp);   /* mod time changed */

2863        if (error) {
2864                VN_RELE(vp);
2865                nfs_rw_exit(&drp->r_rwlock);
2866                return (error);
2867        }

2869        error = geterrno(status);
2870        if (!error) {
2871                if (HAVE_RDDIR_CACHE(drp))
2872                        nfs_purge_rddir_cache(dvp);
2873                if (HAVE_RDDIR_CACHE(VTOR(vp)))
2874                        nfs_purge_rddir_cache(vp);
2875        } else {
2876                PURGE_STALE_FH(error, dvp, cr);
2877                /*
2878                 * System V defines rmdir to return EEXIST, not
2879                 * ENOTEMPTY if the directory is not empty.  Over
2880                 * the wire, the error is NFSERR_ENOTEMPTY which
2881                 * geterrno maps to ENOTEMPTY.
2882                 */
2883                if (error == ENOTEMPTY)
2884                        error = EEXIST;
2885        }

2887        if (error == 0) {
2888                vnevent_rmdir(vp, dvp, nm, ct);
2889        }
2890        VN_RELE(vp);

2892        nfs_rw_exit(&drp->r_rwlock);

2894        return (error);
2895 }

2897 /* ARGSUSED */
2898 static int
2899 nfs_symlink(vnode_t *dvp, char *lnm, struct vattr *tva, char *tnm, cred_t *cr,
```

```
2900         caller_context_t *ct, int flags)
2901 {
2902         int error;
2903         struct nfsslargs args;
2904         enum nfsstat status;
2905         int douprintf;
2906         rnode_t *drp;

2908         if (nfs_zone() != VTOMI(dvp)->mi_zone)
2909                 return (EPERM);
2910         setdiropargs(&args.sla_from, lnm, dvp);
2911         args.sla_sa = &args.sla_sa_buf;
2912         error = vattr_to_sattr(tva, args.sla_sa);
2913         if (error) {
2914                 /* req time field(s) overflow - return immediately */
2915                 return (error);
2916         }
2917         args.sla_tnm = tnm;

2919         drp = VTOR(dvp);
2920         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR(dvp)))
2921                 return (EINTR);

2923         dnlc_remove(dvp, lnm);

2925         douprintf = 1;

2927         error = rfs2call(VTOMI(dvp), RFS_SYMLINK,
2928             xdr_slargs, (caddr_t)&args,
2929             xdr_enum, (caddr_t)&status, cr,
2930             &douprintf, &status, 0, NULL);

2932         PURGE_ATTRCACHE(dvp);   /* mod time changed */

2934         if (!error) {
2935                 error = geterrno(status);
2936                 if (!error) {
2937                         if (HAVE_RDDIR_CACHE(drp))
2938                                 nfs_purge_rddir_cache(dvp);
2939                 } else {
2940                         PURGE_STALE_FH(error, dvp, cr);
2941                 }
2942         }

2944         nfs_rw_exit(&drp->r_rwlock);

2946         return (error);
2947 }

2949 #ifdef DEBUG
2950 static int nfs_readdir_cache_hits = 0;
2951 static int nfs_readdir_cache_shorts = 0;
2952 static int nfs_readdir_cache_waits = 0;
2953 static int nfs_readdir_cache_misses = 0;
2954 static int nfs_readdir_readahead = 0;
2955 #endif

2957 static int nfs_shrinkreaddir = 0;

2959 /*
2960  * Read directory entries.
2961  * There are some weird things to look out for here.  The uio_offset
2962  * field is either 0 or it is the offset returned from a previous
2963  * readdir.  It is an opaque value used by the server to find the
2964  * correct directory block to read. The count field is the number
2965  * of blocks to read on the server.  This is advisory only, the server
```



```
2966  * may return only one block's worth of entries.  Entries may be compressed
2967  * on the server.
2968  */
2969 /* ARGSUSED */
2970 static int
2971 nfs_readdir(vnode_t *vp, struct uio *uiop, cred_t *cr, int *eofp,
2972         caller_context_t *ct, int flags)
2973 {
2974         int error;
2975         size_t count;
2976         rnode_t *rp;
2977         rddir_cache *rdc;
2978         rddir_cache *nrdc;
2979         rddir_cache *rrdc;
2980 #ifdef DEBUG
2981         int missed;
2982 #endif
2983         rddir_cache srdc;
2984         avl_index_t where;

2986         rp = VTOR(vp);

2988         ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));
2989         if (nfs_zone() != VTOMI(vp)->mi_zone)
2990                 return (EIO);
2991         /*
2992          * Make sure that the directory cache is valid.
2993          */
2994         if (HAVE_RDDIR_CACHE(rp)) {
2995                 if (nfs_disable_rddir_cache) {
2996                         /*
2997                          * Setting nfs_disable_rddir_cache in /etc/system
2998                          * allows interoperability with servers that do not
2999                          * properly update the attributes of directories.
3000                          * Any cached information gets purged before an
3001                          * access is made to it.
3002                          */
3003                         nfs_purge_rddir_cache(vp);
3004                 } else {
3005                         error = nfs_validate_caches(vp, cr);
3006                         if (error)
3007                                 return (error);
3008                 }
3009         }

3011         /*
3012          * UGLINESS: SunOS 3.2 servers apparently cannot always handle an
3013          * RFS_READDIR request with rda_count set to more than 0x400. So
3014          * we reduce the request size here purely for compatibility.
3015          *
3016          * In general, this is no longer required.  However, if a server
3017          * is discovered which can not handle requests larger than 1024,
3018          * nfs_shrinkreaddir can be set to 1 to enable this backwards
3019          * compatibility.
3020          *
3021          * In any case, the request size is limited to NFS_MAXDATA bytes.
3022          */
3023         count = MIN(uiop->uio_iov->iov_len,
3024             nfs_shrinkreaddir ? 0x400 : NFS_MAXDATA);

3026         nrdc = NULL;
3027 #ifdef DEBUG
3028         missed = 0;
3029 #endif
3030 top:
3031         /*
```

```
3032          * Short circuit last readdir which always returns 0 bytes.
3033          * This can be done after the directory has been read through
3034          * completely at least once.  This will set r_direof which
3035          * can be used to find the value of the last cookie.
3036          */
3037         mutex_enter(&rp->r_statelock);
3038         if (rp->r_direof != NULL &&
3039             uiop->uio_offset == rp->r_direof->nfs_ncookie) {
3040                 mutex_exit(&rp->r_statelock);
3041 #ifdef DEBUG
3042                 nfs_readdir_cache_shorts++;
3043 #endif
3044                 if (eofp)
3045                         *eofp = 1;
3046                 if (nrdc != NULL)
3047                         rddir_cache_rele(nrdc);
3048                 return (0);
3049         }
3050         /*
3051          * Look for a cache entry.  Cache entries are identified
3052          * by the NFS cookie value and the byte count requested.
3053          */
3054         srdc.nfs_cookie = uiop->uio_offset;
3055         srdc.buflen = count;
3056         rdc = avl_find(&rp->r_dir, &srdc, &where);
3057         if (rdc != NULL) {
3058                 rddir_cache_hold(rdc);
3059                 /*
3060                  * If the cache entry is in the process of being
3061                  * filled in, wait until this completes.  The
3062                  * RDDIRWAIT bit is set to indicate that someone
3063                  * is waiting and then the thread currently
3064                  * filling the entry is done, it should do a
3065                  * cv_broadcast to wakeup all of the threads
3066                  * waiting for it to finish.
3067                  */
3068                 if (rdc->flags & RDDIR) {
3069                         nfs_rw_exit(&rp->r_rwlock);
3070                         rdc->flags |= RDDIRWAIT;
3071 #ifdef DEBUG
3072                         nfs_readdir_cache_waits++;
3073 #endif
3074                         if (!cv_wait_sig(&rdc->cv, &rp->r_statelock)) {
3075                                 /*
3076                                  * We got interrupted, probably
3077                                  * the user typed ^C or an alarm
3078                                  * fired.  We free the new entry
3079                                  * if we allocated one.
3080                                  */
3081                                 mutex_exit(&rp->r_statelock);
3082                                 (void) nfs_rw_enter_sig(&rp->r_rwlock,
3083                                     RW_READER, FALSE);
3084                                 rddir_cache_rele(rdc);
3085                                 if (nrdc != NULL)
3086                                         rddir_cache_rele(nrdc);
3087                                 return (EINTR);
3088                         }
3089                         mutex_exit(&rp->r_statelock);
3090                         (void) nfs_rw_enter_sig(&rp->r_rwlock,
3091                             RW_READER, FALSE);
3092                         rddir_cache_rele(rdc);
3093                         goto top;
3094                 }
3095                 /*
3096                  * Check to see if a readdir is required to
3097                  * fill the entry.  If so, mark this entry
```



```
3098                  * as being filled, remove our reference,
3099                  * and branch to the code to fill the entry.
3100                  */
3101                 if (rdc->flags & RDDIRREQ) {
3102                         rdc->flags &= ~RDDIRREQ;
3103                         rdc->flags |= RDDIR;
3104                         if (nrdc != NULL)
3105                                 rddir_cache_rele(nrdc);
3106                         nrdc = rdc;
3107                         mutex_exit(&rp->r_statelock);
3108                         goto bottom;
3109                 }
3110 #ifdef DEBUG
3111                 if (!missed)
3112                         nfs_readdir_cache_hits++;
3113 #endif
3114                 /*
3115                  * If an error occurred while attempting
3116                  * to fill the cache entry, just return it.
3117                  */
3118                 if (rdc->error) {
3119                         error = rdc->error;
3120                         mutex_exit(&rp->r_statelock);
3121                         rddir_cache_rele(rdc);
3122                         if (nrdc != NULL)
3123                                 rddir_cache_rele(nrdc);
3124                         return (error);
3125                 }

3127                 /*
3128                  * The cache entry is complete and good,
3129                  * copyout the dirent structs to the calling
3130                  * thread.
3131                  */
3132                 error = uiomove(rdc->entries, rdc->entlen, UIO_READ, uiop);

3134                 /*
3135                  * If no error occurred during the copyout,
3136                  * update the offset in the uio struct to
3137                  * contain the value of the next cookie
3138                  * and set the eof value appropriately.
3139                  */
3140                 if (!error) {
3141                         uiop->uio_offset = rdc->nfs_ncookie;
3142                         if (eofp)
3143                                 *eofp = rdc->eof;
3144                 }

3146                 /*
3147                  * Decide whether to do readahead.  Don't if
3148                  * have already read to the end of directory.
3149                  */
3150                 if (rdc->eof) {
3151                         rp->r_direof = rdc;
3152                         mutex_exit(&rp->r_statelock);
3153                         rddir_cache_rele(rdc);
3154                         if (nrdc != NULL)
3155                                 rddir_cache_rele(nrdc);
3156                         return (error);
3157                 }

3159                 /*
3160                  * Check to see whether we found an entry
3161                  * for the readahead.  If so, we don't need
3162                  * to do anything further, so free the new
3163                  * entry if one was allocated.  Otherwise,
```

```
3164                  * allocate a new entry, add it to the cache,
3165                  * and then initiate an asynchronous readdir
3166                  * operation to fill it.
3167                  */
3168                 srdc.nfs_cookie = rdc->nfs_ncookie;
3169                 srdc.buflen = count;
3170                 rrdc = avl_find(&rp->r_dir, &srdc, &where);
3171                 if (rrdc != NULL) {
3172                         if (nrdc != NULL)
3173                                 rddir_cache_rele(nrdc);
3174                 } else {
3175                         if (nrdc != NULL)
3176                                 rrdc = nrdc;
3177                         else {
3178                                 rrdc = rddir_cache_alloc(KM_NOSLEEP);
3179                         }
3180                         if (rrdc != NULL) {
3181                                 rrdc->nfs_cookie = rdc->nfs_ncookie;
3182                                 rrdc->buflen = count;
3183                                 avl_insert(&rp->r_dir, rrdc, where);
3184                                 rddir_cache_hold(rrdc);
3185                                 mutex_exit(&rp->r_statelock);
3186                                 rddir_cache_rele(rdc);
3187 #ifdef DEBUG
3188                                 nfs_readdir_readahead++;
3189 #endif
3190                                 nfs_async_readdir(vp, rrdc, cr, nfsreaddir);
3191                                 return (error);
3192                         }
3193                 }

3195                 mutex_exit(&rp->r_statelock);
3196                 rddir_cache_rele(rdc);
3197                 return (error);
3198         }

3200         /*
3201          * Didn't find an entry in the cache.  Construct a new empty
3202          * entry and link it into the cache.  Other processes attempting
3203          * to access this entry will need to wait until it is filled in.
3204          *
3205          * Since kmem_alloc may block, another pass through the cache
3206          * will need to be taken to make sure that another process
3207          * hasn't already added an entry to the cache for this request.
3208          */
3209         if (nrdc == NULL) {
3210                 mutex_exit(&rp->r_statelock);
3211                 nrdc = rddir_cache_alloc(KM_SLEEP);
3212                 nrdc->nfs_cookie = uiop->uio_offset;
3213                 nrdc->buflen = count;
3214                 goto top;
3215         }

3217         /*
3218          * Add this entry to the cache.
3219          */
3220         avl_insert(&rp->r_dir, nrdc, where);
3221         rddir_cache_hold(nrdc);
3222         mutex_exit(&rp->r_statelock);

3224 bottom:
3225 #ifdef DEBUG
3226         missed = 1;
3227         nfs_readdir_cache_misses++;
3228 #endif
3229         /*
```



```
3230          * Do the readdir.
3231          */
3232         error = nfsreaddir(vp, nrdc, cr);

3234         /*
3235          * If this operation failed, just return the error which occurred.
3236          */
3237         if (error != 0)
3238                 return (error);

3240         /*
3241          * Since the RPC operation will have taken sometime and blocked
3242          * this process, another pass through the cache will need to be
3243          * taken to find the correct cache entry.  It is possible that
3244          * the correct cache entry will not be there (although one was
3245          * added) because the directory changed during the RPC operation
3246          * and the readdir cache was flushed.  In this case, just start
3247          * over.  It is hoped that this will not happen too often... :-)
3248          */
3249         nrdc = NULL;
3250         goto top;
3251         /* NOTREACHED */
3252 }

3254 static int
3255 nfsreaddir(vnode_t *vp, rddir_cache *rdc, cred_t *cr)
3256 {
3257         int error;
3258         struct nfsrddirargs rda;
3259         struct nfsrddirres rd;
3260         rnode_t *rp;
3261         mntinfo_t *mi;
3262         uint_t count;
3263         int douprintf;
3264         failinfo_t fi, *fip;

3266         ASSERT(nfs_zone() == VTOMI(vp)->mi_zone);
3267         count = rdc->buflen;

3269         rp = VTOR(vp);
3270         mi = VTOMI(vp);

3272         rda.rda_fh = *VTOFH(vp);
3273         rda.rda_offset = rdc->nfs_cookie;

3275         /*
3276          * NFS client failover support
3277          * suppress failover unless we have a zero cookie
3278          */
3279         if (rdc->nfs_cookie == (off_t)0) {
3280                 fi.vp = vp;
3281                 fi.fhp = (caddr_t)&rda.rda_fh;
3282                 fi.copyproc = nfscopyfh;
3283                 fi.lookupproc = nfslookup;
3284                 fi.xattrdirproc = acl_getxattrdir2;
3285                 fip = &fi;
3286         } else {
3287                 fip = NULL;
3288         }

3290         rd.rd_entries = kmem_alloc(rdc->buflen, KM_SLEEP);
3291         rd.rd_size = count;
3292         rd.rd_offset = rda.rda_offset;

3294         douprintf = 1;
```

```
3296         if (mi->mi_io_kstats) {
3297                 mutex_enter(&mi->mi_lock);
3298                 kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
3299                 mutex_exit(&mi->mi_lock);
3300         }

3302         do {
3303                 rda.rda_count = MIN(count, mi->mi_curread);
3304                 error = rfs2call(mi, RFS_READDIR,
3305                     xdr_rddirargs, (caddr_t)&rda,
3306                     xdr_getrddirres, (caddr_t)&rd, cr,
3307                     &douprintf, &rd.rd_status, 0, fip);
3308         } while (error == ENFS_TRYAGAIN);

3310         if (mi->mi_io_kstats) {
3311                 mutex_enter(&mi->mi_lock);
3312                 kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
3313                 mutex_exit(&mi->mi_lock);
3314         }

3316         /*
3317          * Since we are actually doing a READDIR RPC, we must have
3318          * exclusive access to the cache entry being filled.  Thus,
3319          * it is safe to update all fields except for the flags
3320          * field.  The r_statelock in the rnode must be held to
3321          * prevent two different threads from simultaneously
3322          * attempting to update the flags field.  This can happen
3323          * if we are turning off RDDIR and the other thread is
3324          * trying to set RDDIRWAIT.
3325          */
3326         ASSERT(rdc->flags & RDDIR);
3327         if (!error) {
3328                 error = geterrno(rd.rd_status);
3329                 if (!error) {
3330                         rdc->nfs_ncookie = rd.rd_offset;
3331                         rdc->eof = rd.rd_eof ? 1 : 0;
3332                         rdc->entlen = rd.rd_size;
3333                         ASSERT(rdc->entlen <= rdc->buflen);
3334 #ifdef DEBUG
3335                         rdc->entries = rddir_cache_buf_alloc(rdc->buflen,
3336                             KM_SLEEP);
3337 #else
3338                         rdc->entries = kmem_alloc(rdc->buflen, KM_SLEEP);
3339 #endif
3340                         bcopy(rd.rd_entries, rdc->entries, rdc->entlen);
3341                         rdc->error = 0;
3342                         if (mi->mi_io_kstats) {
3343                                 mutex_enter(&mi->mi_lock);
3344                                 KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
3345                                 KSTAT_IO_PTR(mi->mi_io_kstats)->nread +=
3346                                     rd.rd_size;
3347                                 mutex_exit(&mi->mi_lock);
3348                         }
3349                 } else {
3350                         PURGE_STALE_FH(error, vp, cr);
3351                 }
3352         }
3353         if (error) {
3354                 rdc->entries = NULL;
3355                 rdc->error = error;
3356         }
3357         kmem_free(rd.rd_entries, rdc->buflen);

3359         mutex_enter(&rp->r_statelock);
3360         rdc->flags &= ~RDDIR;
3361         if (rdc->flags & RDDIRWAIT) {
```



```
3362                 rdc->flags &= ~RDDIRWAIT;
3363                 cv_broadcast(&rdc->cv);
3364         }
3365         if (error)
3366                 rdc->flags |= RDDIRREQ;
3367         mutex_exit(&rp->r_statelock);

3369         rddir_cache_rele(rdc);

3371         return (error);
3372 }

3374 #ifdef DEBUG
3375 static int nfs_bio_do_stop = 0;
3376 #endif

3378 static int
3379 nfs_bio(struct buf *bp, cred_t *cr)
3380 {
3381         rnode_t *rp = VTOR(bp->b_vp);
3382         int count;
3383         int error;
3384         cred_t *cred;
3385         uint_t offset;

3387         DTRACE_IO1(start, struct buf *, bp);

3389         ASSERT(nfs_zone() == VTOMI(bp->b_vp)->mi_zone);
3390         offset = dbtob(bp->b_blkno);

3392         if (bp->b_flags & B_READ) {
3393                 mutex_enter(&rp->r_statelock);
3394                 if (rp->r_cred != NULL) {
3395                         cred = rp->r_cred;
3396                         crhold(cred);
3397                 } else {
3398                         rp->r_cred = cr;
3399                         crhold(cr);
3400                         cred = cr;
3401                         crhold(cred);
3402                 }
3403                 mutex_exit(&rp->r_statelock);
3404         read_again:
3405                 error = bp->b_error = nfsread(bp->b_vp, bp->b_un.b_addr,
3406                     offset, bp->b_bcount, &bp->b_resid, cred);

3408                 crfree(cred);
3409                 if (!error) {
3410                         if (bp->b_resid) {
3411                                 /*
3412                                  * Didn't get it all because we hit EOF,
3413                                  * zero all the memory beyond the EOF.
3414                                  */
3415                                 /* bzero(rdaddr + */
3416                                 bzero(bp->b_un.b_addr +
3417                                     bp->b_bcount - bp->b_resid, bp->b_resid);
3418                         }
3419                         mutex_enter(&rp->r_statelock);
3420                         if (bp->b_resid == bp->b_bcount &&
3421                             offset >= rp->r_size) {
3422                                 /*
3423                                  * We didn't read anything at all as we are
3424                                  * past EOF.  Return an error indicator back
3425                                  * but don't destroy the pages (yet).
3426                                  */
3427                                 error = NFS_EOF;
```

```
3428                         }
3429                         mutex_exit(&rp->r_statelock);
3430                 } else if (error == EACCES) {
3431                         mutex_enter(&rp->r_statelock);
3432                         if (cred != cr) {
3433                                 if (rp->r_cred != NULL)
3434                                         crfree(rp->r_cred);
3435                                 rp->r_cred = cr;
3436                                 crhold(cr);
3437                                 cred = cr;
3438                                 crhold(cred);
3439                                 mutex_exit(&rp->r_statelock);
3440                                 goto read_again;
3441                         }
3442                         mutex_exit(&rp->r_statelock);
3443                 }
3444         } else {
3445                 if (!(rp->r_flags & RSTALE)) {
3446                         mutex_enter(&rp->r_statelock);
3447                         if (rp->r_cred != NULL) {
3448                                 cred = rp->r_cred;
3449                                 crhold(cred);
3450                         } else {
3451                                 rp->r_cred = cr;
3452                                 crhold(cr);
3453                                 cred = cr;
3454                                 crhold(cred);
3455                         }
3456                         mutex_exit(&rp->r_statelock);
3457                 write_again:
3458                         mutex_enter(&rp->r_statelock);
3459                         count = MIN(bp->b_bcount, rp->r_size - offset);
3460                         mutex_exit(&rp->r_statelock);
3461                         if (count < 0)
3462                                 cmn_err(CE_PANIC, "nfs_bio: write count < 0");
3463 #ifdef DEBUG
3464                         if (count == 0) {
3465                                 zcmn_err(getzoneid(), CE_WARN,
3466                                     "nfs_bio: zero length write at %d",
3467                                     offset);
3468                                 nfs_printfhandle(&rp->r_fh);
3469                                 if (nfs_bio_do_stop)
3470                                         debug_enter("nfs_bio");
3471                         }
3472 #endif
3473                         error = nfswrite(bp->b_vp, bp->b_un.b_addr, offset,
3474                             count, cred);
3475                         if (error == EACCES) {
3476                                 mutex_enter(&rp->r_statelock);
3477                                 if (cred != cr) {
3478                                         if (rp->r_cred != NULL)
3479                                                 crfree(rp->r_cred);
3480                                         rp->r_cred = cr;
3481                                         crhold(cr);
3482                                         crfree(cred);
3483                                         cred = cr;
3484                                         crhold(cred);
3485                                         mutex_exit(&rp->r_statelock);
3486                                         goto write_again;
3487                                 }
3488                                 mutex_exit(&rp->r_statelock);
3489                         }
3490                         bp->b_error = error;
3491                         if (error && error != EINTR) {
3492                                 /*
3493                                  * Don't print EDQUOT errors on the console.
```



```
3494                                  * Don't print asynchronous EACCES errors.
3495                                  * Don't print EFBIG errors.
3496                                  * Print all other write errors.
3497                                  */
3498                                 if (error != EDQUOT && error != EFBIG &&
3499                                     (error != EACCES ||
3500                                     !(bp->b_flags & B_ASYNC)))
3501                                         nfs_write_error(bp->b_vp, error, cred);
3502                                 /*
3503                                  * Update r_error and r_flags as appropriate.
3504                                  * If the error was ESTALE, then mark the
3505                                  * rnode as not being writeable and save
3506                                  * the error status.  Otherwise, save any
3507                                  * errors which occur from asynchronous
3508                                  * page invalidations.  Any errors occurring
3509                                  * from other operations should be saved
3510                                  * by the caller.
3511                                  */
3512                                 mutex_enter(&rp->r_statelock);
3513                                 if (error == ESTALE) {
3514                                         rp->r_flags |= RSTALE;
3515                                         if (!rp->r_error)
3516                                                 rp->r_error = error;
3517                                 } else if (!rp->r_error &&
3518                                     (bp->b_flags &
3519                                     (B_INVAL|B_FORCE|B_ASYNC)) ==
3520                                     (B_INVAL|B_FORCE|B_ASYNC)) {
3521                                         rp->r_error = error;
3522                                 }
3523                                 mutex_exit(&rp->r_statelock);
3524                         }
3525                         crfree(cred);
3526                 } else {
3527                         error = rp->r_error;
3528                         /*
3529                          * A close may have cleared r_error, if so,
3530                          * propagate ESTALE error return properly
3531                          */
3532                         if (error == 0)
3533                                 error = ESTALE;
3534                 }
3535         }

3537         if (error != 0 && error != NFS_EOF)
3538                 bp->b_flags |= B_ERROR;

3540         DTRACE_IO1(done, struct buf *, bp);

3542         return (error);
3543 }

3545 /* ARGSUSED */
3546 static int
3547 nfs_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
3548 {
3549         struct nfs_fid *fp;
3550         rnode_t *rp;

3552         rp = VTOR(vp);

3554         if (fidp->fid_len < (sizeof (struct nfs_fid) - sizeof (short))) {
3555                 fidp->fid_len = sizeof (struct nfs_fid) - sizeof (short);
3556                 return (ENOSPC);
3557         }
3558         fp = (struct nfs_fid *)fidp;
3559         fp->nf_pad = 0;
```

```
3560         fp->nf_len = sizeof (struct nfs_fid) - sizeof (short);
3561         bcopy(rp->r_fh.fh_buf, fp->nf_data, NFS_FHSIZE);
3562         return (0);
3563 }

3565 /* ARGSUSED2 */
3566 static int
3567 nfs_rwlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
3568 {
3569         rnode_t *rp = VTOR(vp);

3571         if (!write_lock) {
3572                 (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
3573                 return (V_WRITELOCK_FALSE);
3574         }

3576         if ((rp->r_flags & RDIRECTIO) || (VTOMI(vp)->mi_flags & MI_DIRECTIO)) {
3577                 (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
3578                 if (rp->r_mapcnt == 0 && !vn_has_cached_data(vp))
3579                         return (V_WRITELOCK_FALSE);
3580                 nfs_rw_exit(&rp->r_rwlock);
3581         }

3583         (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, FALSE);
3584         return (V_WRITELOCK_TRUE);
3585 }

3587 /* ARGSUSED */
3588 static void
3589 nfs_rwunlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
3590 {
3591         rnode_t *rp = VTOR(vp);

3593         nfs_rw_exit(&rp->r_rwlock);
3594 }

3596 /* ARGSUSED */
3597 static int
3598 nfs_seek(vnode_t *vp, offset_t ooff, offset_t *noffp, caller_context_t *ct)
3599 {

3601         /*
3602          * Because we stuff the readdir cookie into the offset field
3603          * someone may attempt to do an lseek with the cookie which
3604          * we want to succeed.
3605          */
3606         if (vp->v_type == VDIR)
3607                 return (0);
3608         if (*noffp < 0 || *noffp > MAXOFF32_T)
3609                 return (EINVAL);
3610         return (0);
3611 }

3613 /*
3614  * number of NFS_MAXDATA blocks to read ahead
3615  * optimized for 100 base-T.
3616  */
3617 static int nfs_nra = 4;

3619 #ifdef DEBUG
3620 static int nfs_lostpage = 0;    /* number of times we lost original page */
3621 #endif

3623 /*
3624  * Return all the pages from [off..off+len) in file
3625  */
```



```
3626 /* ARGSUSED */
3627 static int
3628 nfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
3629         page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3630         enum seg_rw rw, cred_t *cr, caller_context_t *ct)
3631 {
3632         rnode_t *rp;
3633         int error;
3634         mntinfo_t *mi;

3636         if (vp->v_flag & VNOMAP)
3637                 return (ENOSYS);

3639         ASSERT(off <= MAXOFF32_T);
3640         if (nfs_zone() != VTOMI(vp)->mi_zone)
3641                 return (EIO);
3642         if (protp != NULL)
3643                 *protp = PROT_ALL;

3645         /*
3646          * Now valididate that the caches are up to date.
3647          */
3648         error = nfs_validate_caches(vp, cr);
3649         if (error)
3650                 return (error);

3652         rp = VTOR(vp);
3653         mi = VTOMI(vp);
3654 retry:
3655         mutex_enter(&rp->r_statelock);

3657         /*
3658          * Don't create dirty pages faster than they
3659          * can be cleaned so that the system doesn't
3660          * get imbalanced.  If the async queue is
3661          * maxed out, then wait for it to drain before
3662          * creating more dirty pages.  Also, wait for
3663          * any threads doing pagewalks in the vop_getattr
3664          * entry points so that they don't block for
3665          * long periods.
3666          */
3667         if (rw == S_CREATE) {
3668                 while ((mi->mi_max_threads != 0 &&
3669                     rp->r_awcount > 2 * mi->mi_max_threads) ||
3670                     rp->r_gcount > 0)
3671                         cv_wait(&rp->r_cv, &rp->r_statelock);
3672         }

3674         /*
3675          * If we are getting called as a side effect of an nfs_write()
3676          * operation the local file size might not be extended yet.
3677          * In this case we want to be able to return pages of zeroes.
3678          */
3679         if (off + len > rp->r_size + PAGEOFFSET && seg != segkmap) {
3680                 mutex_exit(&rp->r_statelock);
3681                 return (EFAULT);                /* beyond EOF */
3682         }

3684         mutex_exit(&rp->r_statelock);

3686         error = pvn_getpages(nfs_getapage, vp, off, len, protp, pl, plsz,
  30         if (len <= PAGESIZE) {
  31                 error = nfs_getapage(vp, off, len, protp, pl, plsz,
3687             seg, addr, rw, cr);
  33         } else {
  34                 error = pvn_getpages(nfs_getapage, vp, off, len, protp,
```

```
  35                    pl, plsz, seg, addr, rw, cr);
  36         }

3689         switch (error) {
3690         case NFS_EOF:
3691                 nfs_purge_caches(vp, NFS_NOPURGE_DNLC, cr);
3692                 goto retry;
3693         case ESTALE:
3694                 PURGE_STALE_FH(error, vp, cr);
3695         }

3697         return (error);
3698 }

3700 /*
3701  * Called from pvn_getpages to get a particular page.
  50  * Called from pvn_getpages or nfs_getpage to get a particular page.
3702  */
3703 /* ARGSUSED */
3704 static int
3705 nfs_getapage(vnode_t *vp, u_offset_t off, size_t len, uint_t *protp,
3706         page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
3707         enum seg_rw rw, cred_t *cr)
3708 {
3709         rnode_t *rp;
3710         uint_t bsize;
3711         struct buf *bp;
3712         page_t *pp;
3713         u_offset_t lbn;
3714         u_offset_t io_off;
3715         u_offset_t blkoff;
3716         u_offset_t rablkoff;
3717         size_t io_len;
3718         uint_t blksize;
3719         int error;
3720         int readahead;
3721         int readahead_issued = 0;
3722         int ra_window; /* readahead window */
3723         page_t *pagefound;

3725         if (nfs_zone() != VTOMI(vp)->mi_zone)
3726                 return (EIO);
3727         rp = VTOR(vp);
3728         bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);

3730 reread:
3731         bp = NULL;
3732         pp = NULL;
3733         pagefound = NULL;

3735         if (pl != NULL)
3736                 pl[0] = NULL;

3738         error = 0;
3739         lbn = off / bsize;
3740         blkoff = lbn * bsize;

3742         /*
3743          * Queueing up the readahead before doing the synchronous read
3744          * results in a significant increase in read throughput because
3745          * of the increased parallelism between the async threads and
3746          * the process context.
3747          */
3748         if ((off & ((vp->v_vfsp->vfs_bsize) - 1)) == 0 &&
3749             rw != S_CREATE &&
3750             !(vp->v_flag & VNOCACHE)) {
```



```
3751                 mutex_enter(&rp->r_statelock);

3753                 /*
3754                  * Calculate the number of readaheads to do.
3755                  * a) No readaheads at offset = 0.
3756                  * b) Do maximum(nfs_nra) readaheads when the readahead
3757                  *    window is closed.
3758                  * c) Do readaheads between 1 to (nfs_nra - 1) depending
3759                  *    upon how far the readahead window is open or close.
3760                  * d) No readaheads if rp->r_nextr is not within the scope
3761                  *    of the readahead window (random i/o).
3762                  */

3764                 if (off == 0)
3765                         readahead = 0;
3766                 else if (blkoff == rp->r_nextr)
3767                         readahead = nfs_nra;
3768                 else if (rp->r_nextr > blkoff &&
3769                     ((ra_window = (rp->r_nextr - blkoff) / bsize)
3770                     <= (nfs_nra - 1)))
3771                         readahead = nfs_nra - ra_window;
3772                 else
3773                         readahead = 0;

3775                 rablkoff = rp->r_nextr;
3776                 while (readahead > 0 && rablkoff + bsize < rp->r_size) {
3777                         mutex_exit(&rp->r_statelock);
3778                         if (nfs_async_readahead(vp, rablkoff + bsize,
3779                             addr + (rablkoff + bsize - off), seg, cr,
3780                             nfs_readahead) < 0) {
3781                                 mutex_enter(&rp->r_statelock);
3782                                 break;
3783                         }
3784                         readahead--;
3785                         rablkoff += bsize;
3786                         /*
3787                          * Indicate that we did a readahead so
3788                          * readahead offset is not updated
3789                          * by the synchronous read below.
3790                          */
3791                         readahead_issued = 1;
3792                         mutex_enter(&rp->r_statelock);
3793                         /*
3794                          * set readahead offset to
3795                          * offset of last async readahead
3796                          * request.
3797                          */
3798                         rp->r_nextr = rablkoff;
3799                 }
3800                 mutex_exit(&rp->r_statelock);
3801         }

3803 again:
3804         if ((pagefound = page_exists(vp, off)) == NULL) {
3805                 if (pl == NULL) {
3806                         (void) nfs_async_readahead(vp, blkoff, addr, seg, cr,
3807                             nfs_readahead);
3808                 } else if (rw == S_CREATE) {
3809                         /*
3810                          * Block for this page is not allocated, or the offset
3811                          * is beyond the current allocation size, or we're
3812                          * allocating a swap slot and the page was not found,
3813                          * so allocate it and return a zero page.
3814                          */
3815                         if ((pp = page_create_va(vp, off,
3816                             PAGESIZE, PG_WAIT, seg, addr)) == NULL)
```

```
3817                            cmn_err(CE_PANIC, "nfs_getapage: page_create");
3818                    io_len = PAGESIZE;
3819                    mutex_enter(&rp->r_statelock);
3820                    rp->r_nextr = off + PAGESIZE;
3821                    mutex_exit(&rp->r_statelock);
3822            } else {
3823                    /*
3824                     * Need to go to server to get a BLOCK, exception to
3825                     * that being while reading at offset = 0 or doing
3826                     * random i/o, in that case read only a PAGE.
3827                     */
3828                    mutex_enter(&rp->r_statelock);
3829                    if (blkoff < rp->r_size &&
3830                        blkoff + bsize >= rp->r_size) {
3831                            /*
3832                             * If only a block or less is left in
3833                             * the file, read all that is remaining.
3834                             */
3835                            if (rp->r_size <= off) {
3836                                    /*
3837                                     * Trying to access beyond EOF,
3838                                     * set up to get at least one page.
3839                                     */
3840                                    blksize = off + PAGESIZE - blkoff;
3841                            } else
3842                                    blksize = rp->r_size - blkoff;
3843                    } else if ((off == 0) ||
3844                        (off != rp->r_nextr && !readahead_issued)) {
3845                            blksize = PAGESIZE;
3846                            blkoff = off; /* block = page here */
3847                    } else
3848                            blksize = bsize;
3849                    mutex_exit(&rp->r_statelock);

3851                    pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
3852                        &io_len, blkoff, blksize, 0);

3854                    /*
3855                     * Some other thread has entered the page,
3856                     * so just use it.
3857                     */
3858                    if (pp == NULL)
3859                            goto again;

3861                    /*
3862                     * Now round the request size up to page boundaries.
3863                     * This ensures that the entire page will be
3864                     * initialized to zeroes if EOF is encountered.
3865                     */
3866                    io_len = ptob(btopr(io_len));

3868                    bp = pageio_setup(pp, io_len, vp, B_READ);
3869                    ASSERT(bp != NULL);

3871                    /*
3872                     * pageio_setup should have set b_addr to 0.  This
3873                     * is correct since we want to do I/O on a page
3874                     * boundary.  bp_mapin will use this addr to calculate
3875                     * an offset, and then set b_addr to the kernel virtual
3876                     * address it allocated for us.
3877                     */
3878                    ASSERT(bp->b_un.b_addr == 0);

3880                    bp->b_edev = 0;
3881                    bp->b_dev = 0;
3882                    bp->b_lblkno = lbtodb(io_off);
```



```
3883                    bp->b_file = vp;
3884                    bp->b_offset = (offset_t)off;
3885                    bp_mapin(bp);

3887                    /*
3888                     * If doing a write beyond what we believe is EOF,
3889                     * don't bother trying to read the pages from the
3890                     * server, we'll just zero the pages here.  We
3891                     * don't check that the rw flag is S_WRITE here
3892                     * because some implementations may attempt a
3893                     * read access to the buffer before copying data.
3894                     */
3895                    mutex_enter(&rp->r_statelock);
3896                    if (io_off >= rp->r_size && seg == segkmap) {
3897                            mutex_exit(&rp->r_statelock);
3898                            bzero(bp->b_un.b_addr, io_len);
3899                    } else {
3900                            mutex_exit(&rp->r_statelock);
3901                            error = nfs_bio(bp, cr);
3902                    }

3904                    /*
3905                     * Unmap the buffer before freeing it.
3906                     */
3907                    bp_mapout(bp);
3908                    pageio_done(bp);

3910                    if (error == NFS_EOF) {
3911                            /*
3912                             * If doing a write system call just return
3913                             * zeroed pages, else user tried to get pages
3914                             * beyond EOF, return error.  We don't check
3915                             * that the rw flag is S_WRITE here because
3916                             * some implementations may attempt a read
3917                             * access to the buffer before copying data.
3918                             */
3919                            if (seg == segkmap)
3920                                    error = 0;
3921                            else
3922                                    error = EFAULT;
3923                    }

3925                    if (!readahead_issued && !error) {
3926                            mutex_enter(&rp->r_statelock);
3927                            rp->r_nextr = io_off + io_len;
3928                            mutex_exit(&rp->r_statelock);
3929                    }
3930            }
3931    }

3933 out:
3934    if (pl == NULL)
3935            return (error);

3937    if (error) {
3938            if (pp != NULL)
3939                    pvn_read_done(pp, B_ERROR);
3940            return (error);
3941    }

3943    if (pagefound) {
3944            se_t se = (rw == S_CREATE ? SE_EXCL : SE_SHARED);

3946            /*
3947             * Page exists in the cache, acquire the appropriate lock.
3948             * If this fails, start all over again.
```

```
3949                  */
3950                 if ((pp = page_lookup(vp, off, se)) == NULL) {
3951 #ifdef DEBUG
3952                         nfs_lostpage++;
3953 #endif
3954                         goto reread;
3955                 }
3956                 pl[0] = pp;
3957                 pl[1] = NULL;
3958                 return (0);
3959         }

3961         if (pp != NULL)
3962                 pvn_plist_init(pp, pl, plsz, off, io_len, rw);

3964         return (error);
3965 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   58491 Thu Jan  8 09:14:35 2015
new/usr/src/uts/common/fs/pcfs/pc_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/user.h>
#include <sys/buf.h>
#include <sys/stat.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/dirent.h>
#include <sys/vnode.h>
#include <sys/proc.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <sys/uio.h>
#include <sys/fs/pc_label.h>
#include <sys/fs/pc_fs.h>
#include <sys/fs/pc_dir.h>
#include <sys/fs/pc_node.h>
#include <sys/mman.h>
#include <sys/pathname.h>
#include <sys/vmsystm.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/statvfs.h>
#include <sys/unistd.h>
#include <sys/kmem.h>
#include <sys/conf.h>
#include <sys/flock.h>
```



```
#include <sys/policy.h>
#include <sys/sdt.h>
#include <sys/sunddi.h>
#include <sys/types.h>
#include <sys/errno.h>

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

#include <fs/fs_subr.h>

static int pcfs_open(struct vnode **, int, struct cred *, caller_context_t *ct);
static int pcfs_close(struct vnode *, int, int, offset_t, struct cred *,
        caller_context_t *ct);
static int pcfs_read(struct vnode *, struct uio *, int, struct cred *,
        caller_context_t *);
static int pcfs_write(struct vnode *, struct uio *, int, struct cred *,
        caller_context_t *);
static int pcfs_getattr(struct vnode *, struct vattr *, int, struct cred *,
        caller_context_t *ct);
static int pcfs_setattr(struct vnode *, struct vattr *, int, struct cred *,
        caller_context_t *);
static int pcfs_access(struct vnode *, int, int, struct cred *,
        caller_context_t *ct);
static int pcfs_lookup(struct vnode *, char *, struct vnode **,
        struct pathname *, int, struct vnode *, struct cred *,
        caller_context_t *, int *, pathname_t *);
static int pcfs_create(struct vnode *, char *, struct vattr *,
        enum vcexcl, int mode, struct vnode **, struct cred *, int,
        caller_context_t *, vsecattr_t *);
static int pcfs_remove(struct vnode *, char *, struct cred *,
        caller_context_t *, int);
static int pcfs_rename(struct vnode *, char *, struct vnode *, char *,
        struct cred *, caller_context_t *, int);
static int pcfs_mkdir(struct vnode *, char *, struct vattr *, struct vnode **,
        struct cred *, caller_context_t *, int, vsecattr_t *);
static int pcfs_rmdir(struct vnode *, char *, struct vnode *, struct cred *,
        caller_context_t *, int);
static int pcfs_readdir(struct vnode *, struct uio *, struct cred *, int *,
        caller_context_t *, int);
static int pcfs_fsync(struct vnode *, int, struct cred *, caller_context_t *);
static void pcfs_inactive(struct vnode *, struct cred *, caller_context_t *);
static int pcfs_fid(struct vnode *vp, struct fid *fidp, caller_context_t *);
static int pcfs_space(struct vnode *, int, struct flock64 *, int,
        offset_t, cred_t *, caller_context_t *);
static int pcfs_getpage(struct vnode *, offset_t, size_t, uint_t *, page_t *[],
        size_t, struct seg *, caddr_t, enum seg_rw, struct cred *,
        caller_context_t *);
static int pcfs_getapage(struct vnode *, u_offset_t, size_t, uint_t *,
        page_t *[], size_t, struct seg *, caddr_t, enum seg_rw, struct cred *);
static int pcfs_putpage(struct vnode *, offset_t, size_t, int, struct cred *,
        caller_context_t *);
static int pcfs_map(struct vnode *, offset_t, struct as *, caddr_t *, size_t,
        uchar_t, uchar_t, uint_t, struct cred *, caller_context_t *);
static int pcfs_addmap(struct vnode *, offset_t, struct as *, caddr_t,
        size_t, uchar_t, uchar_t, uint_t, struct cred *, caller_context_t *);
static int pcfs_delmap(struct vnode *, offset_t, struct as *, caddr_t,
        size_t, uint_t, uint_t, uint_t, struct cred *, caller_context_t *);
static int pcfs_seek(struct vnode *, offset_t, offset_t *,
        caller_context_t *);
```

```
128 static int pcfs_pathconf(struct vnode *, int, ulong_t *, struct cred *,
129         caller_context_t *);

131 int pcfs_putapage(struct vnode *, page_t *, u_offset_t *, size_t *, int,
132         struct cred *);
133 static int rwpcp(struct pcnode *, struct uio *, enum uio_rw, int);
134 static int get_long_fn_chunk(struct pcdir_lfn *ep, char *buf);

136 extern krwlock_t pcnodes_lock;

138 #define lround(r)       (((r)+sizeof (long long)-1)&(~(sizeof (long long)-1)))

140 /*
141  * vnode op vectors for files and directories.
142  */
143 struct vnodeops *pcfs_fvnodeops;
144 struct vnodeops *pcfs_dvnodeops;

146 const fs_operation_def_t pcfs_fvnodeops_template[] = {
147         VOPNAME_OPEN,           { .vop_open = pcfs_open },
148         VOPNAME_CLOSE,          { .vop_close = pcfs_close },
149         VOPNAME_READ,           { .vop_read = pcfs_read },
150         VOPNAME_WRITE,          { .vop_write = pcfs_write },
151         VOPNAME_GETATTR,        { .vop_getattr = pcfs_getattr },
152         VOPNAME_SETATTR,        { .vop_setattr = pcfs_setattr },
153         VOPNAME_ACCESS,         { .vop_access = pcfs_access },
154         VOPNAME_FSYNC,          { .vop_fsync = pcfs_fsync },
155         VOPNAME_INACTIVE,       { .vop_inactive = pcfs_inactive },
156         VOPNAME_FID,            { .vop_fid = pcfs_fid },
157         VOPNAME_SEEK,           { .vop_seek = pcfs_seek },
158         VOPNAME_SPACE,          { .vop_space = pcfs_space },
159         VOPNAME_GETPAGE,        { .vop_getpage = pcfs_getpage },
160         VOPNAME_PUTPAGE,        { .vop_putpage = pcfs_putpage },
161         VOPNAME_MAP,            { .vop_map = pcfs_map },
162         VOPNAME_ADDMAP,         { .vop_addmap = pcfs_addmap },
163         VOPNAME_DELMAP,         { .vop_delmap = pcfs_delmap },
164         VOPNAME_PATHCONF,       { .vop_pathconf = pcfs_pathconf },
165         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
166         NULL,                   NULL
167 };

169 const fs_operation_def_t pcfs_dvnodeops_template[] = {
170         VOPNAME_OPEN,           { .vop_open = pcfs_open },
171         VOPNAME_CLOSE,          { .vop_close = pcfs_close },
172         VOPNAME_GETATTR,        { .vop_getattr = pcfs_getattr },
173         VOPNAME_SETATTR,        { .vop_setattr = pcfs_setattr },
174         VOPNAME_ACCESS,         { .vop_access = pcfs_access },
175         VOPNAME_LOOKUP,         { .vop_lookup = pcfs_lookup },
176         VOPNAME_CREATE,         { .vop_create = pcfs_create },
177         VOPNAME_REMOVE,         { .vop_remove = pcfs_remove },
178         VOPNAME_RENAME,         { .vop_rename = pcfs_rename },
179         VOPNAME_MKDIR,          { .vop_mkdir = pcfs_mkdir },
180         VOPNAME_RMDIR,          { .vop_rmdir = pcfs_rmdir },
181         VOPNAME_READDIR,        { .vop_readdir = pcfs_readdir },
182         VOPNAME_FSYNC,          { .vop_fsync = pcfs_fsync },
183         VOPNAME_INACTIVE,       { .vop_inactive = pcfs_inactive },
184         VOPNAME_FID,            { .vop_fid = pcfs_fid },
185         VOPNAME_SEEK,           { .vop_seek = pcfs_seek },
186         VOPNAME_PATHCONF,       { .vop_pathconf = pcfs_pathconf },
187         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
188         NULL,                   NULL
189 };


192 /*ARGSUSED*/
193 static int
```



```
194 pcfs_open(
195         struct vnode **vpp,
196         int flag,
197         struct cred *cr,
198         caller_context_t *ct)
199 {
200         return (0);
201 }

203 /*
204  * files are sync'ed on close to keep floppy up to date
205  */

207 /*ARGSUSED*/
208 static int
209 pcfs_close(
210         struct vnode *vp,
211         int flag,
212         int count,
213         offset_t offset,
214         struct cred *cr,
215         caller_context_t *ct)
216 {
217         return (0);
218 }

220 /*ARGSUSED*/
221 static int
222 pcfs_read(
223         struct vnode *vp,
224         struct uio *uiop,
225         int ioflag,
226         struct cred *cr,
227         struct caller_context *ct)
228 {
229         struct pcfs *fsp;
230         struct pcnode *pcp;
231         int error;

233         fsp = VFSTOPCFS(vp->v_vfsp);
234         if (error = pc_verify(fsp))
235                 return (error);
236         error = pc_lockfs(fsp, 0, 0);
237         if (error)
238                 return (error);
239         if ((pcp = VTOPC(vp)) == NULL || pcp->pc_flags & PC_INVAL) {
240                 pc_unlockfs(fsp);
241                 return (EIO);
242         }
243         error = rwpcp(pcp, uiop, UIO_READ, ioflag);
244         if ((fsp->pcfs_vfs->vfs_flag & VFS_RDONLY) == 0) {
245                 pc_mark_acc(fsp, pcp);
246         }
247         pc_unlockfs(fsp);
248         if (error) {
249                 PC_DPRINTF1(1, "pcfs_read: io error = %d\n", error);
250         }
251         return (error);
252 }

254 /*ARGSUSED*/
255 static int
256 pcfs_write(
257         struct vnode *vp,
258         struct uio *uiop,
259         int ioflag,
```

```
260         struct cred *cr,
261         struct caller_context *ct)
262 {
263         struct pcfs *fsp;
264         struct pcnode *pcp;
265         int error;

267         fsp = VFSTOPCFS(vp->v_vfsp);
268         if (error = pc_verify(fsp))
269                 return (error);
270         error = pc_lockfs(fsp, 0, 0);
271         if (error)
272                 return (error);
273         if ((pcp = VTOPC(vp)) == NULL || pcp->pc_flags & PC_INVAL) {
274                 pc_unlockfs(fsp);
275                 return (EIO);
276         }
277         if (ioflag & FAPPEND) {
278                 /*
279                  * in append mode start at end of file.
280                  */
281                 uiop->uio_loffset = pcp->pc_size;
282         }
283         error = rwpcp(pcp, uiop, UIO_WRITE, ioflag);
284         pcp->pc_flags |= PC_MOD;
285         pc_mark_mod(fsp, pcp);
286         if (ioflag & (FSYNC|FDSYNC))
287                 (void) pc_nodeupdate(pcp);

289         pc_unlockfs(fsp);
290         if (error) {
291                 PC_DPRINTF1(1, "pcfs_write: io error = %d\n", error);
292         }
293         return (error);
294 }

296 /*
297  * read or write a vnode
298  */
299 static int
300 rwpcp(
301         struct pcnode *pcp,
302         struct uio *uio,
303         enum uio_rw rw,
304         int ioflag)
305 {
306         struct vnode *vp = PCTOV(pcp);
307         struct pcfs *fsp;
308         daddr_t bn;                     /* phys block number */
309         int n;
310         offset_t off;
311         caddr_t base;
312         int mapon, pagecreate;
313         int newpage;
314         int error = 0;
315         rlim64_t limit = uio->uio_llimit;
316         int oresid = uio->uio_resid;

318         /*
319          * If the filesystem was umounted by force, return immediately.
320          */
321         if (vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
322                 return (EIO);

324         PC_DPRINTF4(5, "rwpcp pcp=%p off=%lld resid=%ld size=%u\n", (void *)pcp,
325             uio->uio_loffset, uio->uio_resid, pcp->pc_size);
```



```
327         ASSERT(rw == UIO_READ || rw == UIO_WRITE);
328         ASSERT(vp->v_type == VREG);

330         if (uio->uio_loffset >= UINT32_MAX && rw == UIO_READ) {
331                 return (0);
332         }

334         if (uio->uio_loffset < 0)
335                 return (EINVAL);

337         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
338                 limit = MAXOFFSET_T;

340         if (uio->uio_loffset >= limit && rw == UIO_WRITE) {
341                 proc_t *p = ttoproc(curthread);

343                 mutex_enter(&p->p_lock);
344                 (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE], p->p_rctls,
345                     p, RCA_UNSAFE_SIGINFO);
346                 mutex_exit(&p->p_lock);
347                 return (EFBIG);
348         }

350         /* the following condition will occur only for write */

352         if (uio->uio_loffset >= UINT32_MAX)
353                 return (EFBIG);

355         if (uio->uio_resid == 0)
356                 return (0);

358         if (limit > UINT32_MAX)
359                 limit = UINT32_MAX;

361         fsp = VFSTOPCFS(vp->v_vfsp);
362         if (fsp->pcfs_flags & PCFS_IRRECOV)
363                 return (EIO);

365         do {
366                 /*
367                  * Assignments to "n" in this block may appear
368                  * to overflow in some cases.  However, after careful
369                  * analysis it was determined that all assignments to
370                  * "n" serve only to make "n" smaller.  Since "n"
371                  * starts out as no larger than MAXBSIZE, "int" is
372                  * safe.
373                  */
374                 off = uio->uio_loffset & MAXBMASK;
375                 mapon = (int)(uio->uio_loffset & MAXBOFFSET);
376                 n = MIN(MAXBSIZE - mapon, uio->uio_resid);
377                 if (rw == UIO_READ) {
378                         offset_t diff;

380                         diff = pcp->pc_size - uio->uio_loffset;
381                         if (diff <= 0)
382                                 return (0);
383                         if (diff < n)
384                                 n = (int)diff;
385                 }
386                 /*
387                  * Compare limit with the actual offset + n, not the
388                  * rounded down offset "off" or we will overflow
389                  * the maximum file size after all.
390                  */
391                 if (rw == UIO_WRITE && uio->uio_loffset + n >= limit) {
```

```
392                         if (uio->uio_loffset >= limit) {
393                                 error = EFBIG;
394                                 break;
395                         }
396                         n = (int)(limit - uio->uio_loffset);
397                 }

399                 /*
400                  * Touch the page and fault it in if it is not in
401                  * core before segmap_getmapflt can lock it. This
402                  * is to avoid the deadlock if the buffer is mapped
403                  * to the same file through mmap which we want to
404                  * write to.
405                  */
406                 uio_prefaultpages((long)n, uio);

408                 base = segmap_getmap(segkmap, vp, (u_offset_t)off);
409                 pagecreate = 0;
410                 newpage = 0;
411                 if (rw == UIO_WRITE) {
412                         /*
413                          * If PAGESIZE < MAXBSIZE, perhaps we ought to deal
414                          * with one page at a time, instead of one MAXBSIZE
415                          * at a time, so we can fully explore pagecreate
416                          * optimization??
417                          */
418                         if (uio->uio_loffset + n > pcp->pc_size) {
419                                 uint_t ncl, lcn;

421                                 ncl = (uint_t)howmany((offset_t)pcp->pc_size,
422                                     fsp->pcfs_clsize);
423                                 if (uio->uio_loffset > pcp->pc_size &&
424                                     ncl < (uint_t)howmany(uio->uio_loffset,
425                                     fsp->pcfs_clsize)) {
426                                         /*
427                                          * Allocate and zerofill skipped
428                                          * clusters. This may not be worth the
429                                          * effort since a small lseek beyond
430                                          * eof but still within the cluster
431                                          * will not be zeroed out.
432                                          */
433                                         lcn = pc_lblkno(fsp, uio->uio_loffset);
434                                         error = pc_balloc(pcp, (daddr_t)lcn,
435                                             1, &bn);
436                                         ncl = lcn + 1;
437                                 }
438                                 if (!error &&
439                                     ncl < (uint_t)howmany(uio->uio_loffset + n,
440                                     fsp->pcfs_clsize))
441                                         /*
442                                          * allocate clusters w/o zerofill
443                                          */
444                                         error = pc_balloc(pcp,
445                                             (daddr_t)pc_lblkno(fsp,
446                                             uio->uio_loffset + n - 1),
447                                             0, &bn);

449                                 pcp->pc_flags |= PC_CHG;

451                                 if (error) {
452                                         pc_cluster32_t ncl;
453                                         int nerror;

455                                         /*
456                                          * figure out new file size from
457                                          * cluster chain length. If this
```



```
458                                          * is detected to loop, the chain
459                                          * is corrupted and we'd better
460                                          * keep our fingers off that file.
461                                          */
462                                         nerror = pc_fileclsize(fsp,
463                                             pcp->pc_scluster, &ncl);
464                                         if (nerror) {
465                                                 PC_DPRINTF1(2,
466                                                     "cluster chain "
467                                                     "corruption, "
468                                                     "scluster=%d\n",
469                                                     pcp->pc_scluster);
470                                                 pcp->pc_size = 0;
471                                                 pcp->pc_flags |= PC_INVAL;
472                                                 error = nerror;
473                                                 (void) segmap_release(segkmap,
474                                                     base, 0);
475                                                 break;
476                                         }
477                                         pcp->pc_size = fsp->pcfs_clsize * ncl;

479                                         if (error == ENOSPC &&
480                                             (pcp->pc_size - uio->uio_loffset)
481                                             > 0) {
482                                                 PC_DPRINTF3(2, "rwpcp ENOSPC "
483                                                     "off=%lld n=%d size=%d\n",
484                                                     uio->uio_loffset,
485                                                     n, pcp->pc_size);
486                                                 n = (int)(pcp->pc_size -
487                                                     uio->uio_loffset);
488                                         } else {
489                                                 PC_DPRINTF1(1,
490                                                     "rwpcp error1=%d\n", error);
491                                                 (void) segmap_release(segkmap,
492                                                     base, 0);
493                                                 break;
494                                         }
495                                 } else {
496                                         pcp->pc_size =
497                                             (uint_t)(uio->uio_loffset + n);
498                                 }
499                                 if (mapon == 0) {
500                                         newpage = segmap_pagecreate(segkmap,
501                                             base, (size_t)n, 0);
502                                         pagecreate = 1;
503                                 }
504                         } else if (n == MAXBSIZE) {
505                                 newpage = segmap_pagecreate(segkmap, base,
506                                     (size_t)n, 0);
507                                 pagecreate = 1;
508                         }
509                 }
510                 error = uiomove(base + mapon, (size_t)n, rw, uio);

512                 if (pagecreate && uio->uio_loffset <
513                     roundup(off + mapon + n, PAGESIZE)) {
514                         offset_t nzero, nmoved;

516                         nmoved = uio->uio_loffset - (off + mapon);
517                         nzero = roundup(mapon + n, PAGESIZE) - nmoved;
518                         (void) kzero(base + mapon + nmoved, (size_t)nzero);
519                 }

521                 /*
522                  * Unlock the pages which have been allocated by
523                  * page_create_va() in segmap_pagecreate().
```

```
524                  */
525                 if (newpage) {
526                         segmap_pageunlock(segkmap, base, (size_t)n,
527                             rw == UIO_WRITE ? S_WRITE : S_READ);
528                 }

530                 if (error) {
531                         PC_DPRINTF1(1, "rwpcp error2=%d\n", error);
532                         /*
533                          * If we failed on a write, we may have already
534                          * allocated file blocks as well as pages.  It's hard
535                          * to undo the block allocation, but we must be sure
536                          * to invalidate any pages that may have been
537                          * allocated.
538                          */
539                         if (rw == UIO_WRITE)
540                                 (void) segmap_release(segkmap, base, SM_INVAL);
541                         else
542                                 (void) segmap_release(segkmap, base, 0);
543                 } else {
544                         uint_t flags = 0;

546                         if (rw == UIO_READ) {
547                                 if (n + mapon == MAXBSIZE ||
548                                     uio->uio_loffset == pcp->pc_size)
549                                         flags = SM_DONTNEED;
550                         } else if (ioflag & (FSYNC|FDSYNC)) {
551                                 flags = SM_WRITE;
552                         } else if (n + mapon == MAXBSIZE) {
553                                 flags = SM_WRITE|SM_ASYNC|SM_DONTNEED;
554                         }
555                         error = segmap_release(segkmap, base, flags);
556                 }

558         } while (error == 0 && uio->uio_resid > 0 && n != 0);

560         if (oresid != uio->uio_resid)
561                 error = 0;
562         return (error);
563 }

565 /*ARGSUSED*/
566 static int
567 pcfs_getattr(
568         struct vnode *vp,
569         struct vattr *vap,
570         int flags,
571         struct cred *cr,
572         caller_context_t *ct)
573 {
574         struct pcnode *pcp;
575         struct pcfs *fsp;
576         int error;
577         char attr;
578         struct pctime atime;
579         int64_t unixtime;

581         PC_DPRINTF1(8, "pcfs_getattr: vp=%p\n", (void *)vp);

583         fsp = VFSTOPCFS(vp->v_vfsp);
584         error = pc_lockfs(fsp, 0, 0);
585         if (error)
586                 return (error);

588         /*
589          * Note that we don't check for "invalid node" (PC_INVAL) here
```



```
590          * only in order to make stat() succeed. We allow no I/O on such
591          * a node, but do allow to check for its existence.
592          */
593         if ((pcp = VTOPC(vp)) == NULL) {
594                 pc_unlockfs(fsp);
595                 return (EIO);
596         }
597         /*
598          * Copy from pcnode.
599          */
600         vap->va_type = vp->v_type;
601         attr = pcp->pc_entry.pcd_attr;
602         if (PCA_IS_HIDDEN(fsp, attr))
603                 vap->va_mode = 0;
604         else if (attr & PCA_LABEL)
605                 vap->va_mode = 0444;
606         else if (attr & PCA_RDONLY)
607                 vap->va_mode = 0555;
608         else if (fsp->pcfs_flags & PCFS_BOOTPART) {
609                 vap->va_mode = 0755;
610         } else {
611                 vap->va_mode = 0777;
612         }

614         if (attr & PCA_DIR)
615                 vap->va_mode |= S_IFDIR;
616         else
617                 vap->va_mode |= S_IFREG;
618         if (fsp->pcfs_flags & PCFS_BOOTPART) {
619                 vap->va_uid = 0;
620                 vap->va_gid = 0;
621         } else {
622                 vap->va_uid = crgetuid(cr);
623                 vap->va_gid = crgetgid(cr);
624         }
625         vap->va_fsid = vp->v_vfsp->vfs_dev;
626         vap->va_nodeid = (ino64_t)pc_makenodeid(pcp->pc_eblkno,
627             pcp->pc_eoffset, pcp->pc_entry.pcd_attr,
628             pc_getstartcluster(fsp, &pcp->pc_entry), pc_direntpersec(fsp));
629         vap->va_nlink = 1;
630         vap->va_size = (u_offset_t)pcp->pc_size;
631         vap->va_rdev = 0;
632         vap->va_nblocks =
633             (fsblkcnt64_t)howmany((offset_t)pcp->pc_size, DEV_BSIZE);
634         vap->va_blksize = fsp->pcfs_clsize;

636         /*
637          * FAT root directories have no timestamps. In order not to return
638          * "time zero" (1/1/1970), we record the time of the mount and give
639          * that. This breaks less expectations.
640          */
641         if (vp->v_flag & VROOT) {
642                 vap->va_mtime = fsp->pcfs_mounttime;
643                 vap->va_atime = fsp->pcfs_mounttime;
644                 vap->va_ctime = fsp->pcfs_mounttime;
645                 pc_unlockfs(fsp);
646                 return (0);
647         }

649         pc_pcttotv(&pcp->pc_entry.pcd_mtime, &unixtime);
650         if ((fsp->pcfs_flags & PCFS_NOCLAMPTIME) == 0) {
651                 if (unixtime > INT32_MAX)
652                         DTRACE_PROBE1(pcfs__mtimeclamped, int64_t, unixtime);
653                 unixtime = MIN(unixtime, INT32_MAX);
654         } else if (unixtime > INT32_MAX &&
655             get_udatamodel() == DATAMODEL_ILP32) {
```

```
656                 pc_unlockfs(fsp);
657                 DTRACE_PROBE1(pcfs__mtimeoverflowed, int64_t, unixtime);
658                 return (EOVERFLOW);
659         }
660
661         vap->va_mtime.tv_sec = (time_t)unixtime;
662         vap->va_mtime.tv_nsec = 0;
663
664         /*
665          * FAT doesn't know about POSIX ctime.
666          * Best approximation is to always set it to mtime.
667          */
668         vap->va_ctime = vap->va_mtime;
669
670         /*
671          * FAT only stores "last access date". If that's the
672          * same as the date of last modification then the time
673          * of last access is known. Otherwise, use midnight.
674          */
675         atime.pct_date = pcp->pc_entry.pcd_ladate;
676         if (atime.pct_date == pcp->pc_entry.pcd_mtime.pct_date)
677                 atime.pct_time = pcp->pc_entry.pcd_mtime.pct_time;
678         else
679                 atime.pct_time = 0;
680         pc_pcttotv(&atime, &unixtime);
681         if ((fsp->pcfs_flags & PCFS_NOCLAMPTIME) == 0) {
682                 if (unixtime > INT32_MAX)
683                         DTRACE_PROBE1(pcfs__atimeclamped, int64_t, unixtime);
684                 unixtime = MIN(unixtime, INT32_MAX);
685         } else if (unixtime > INT32_MAX &&
686             get_udatamodel() == DATAMODEL_ILP32) {
687                 pc_unlockfs(fsp);
688                 DTRACE_PROBE1(pcfs__atimeoverflowed, int64_t, unixtime);
689                 return (EOVERFLOW);
690         }
691
692         vap->va_atime.tv_sec = (time_t)unixtime;
693         vap->va_atime.tv_nsec = 0;
694
695         pc_unlockfs(fsp);
696         return (0);
697 }
698
699
700 /*ARGSUSED*/
701 static int
702 pcfs_setattr(
703         struct vnode *vp,
704         struct vattr *vap,
705         int flags,
706         struct cred *cr,
707         caller_context_t *ct)
708 {
709         struct pcnode *pcp;
710         mode_t mask = vap->va_mask;
711         int error;
712         struct pcfs *fsp;
713         timestruc_t now, *timep;
714
715         PC_DPRINTF2(6, "pcfs_setattr: vp=%p mask=%x\n", (void *)vp, (int)mask);
716         /*
717          * cannot set these attributes
718          */
719         if (mask & (AT_NOSET | AT_UID | AT_GID)) {
720                 return (EINVAL);
721         }
```



```
722         /*
723          * pcfs_setattr is now allowed on directories to avoid silly warnings
724          * from 'tar' when it tries to set times on a directory, and console
725          * printf's on the NFS server when it gets EINVAL back on such a
726          * request. One possible problem with that since a directory entry
727          * identifies a file, '.' and all the '..' entries in subdirectories
728          * may get out of sync when the directory is updated since they're
729          * treated like separate files. We could fix that by looking for
730          * '.' and giving it the same attributes, and then looking for
731          * all the subdirectories and updating '..', but that's pretty
732          * expensive for something that doesn't seem likely to matter.
733          */
734         /* can't do some ops on directories anyway */
735         if ((vp->v_type == VDIR) &&
736             (mask & AT_SIZE)) {
737                 return (EINVAL);
738         }
739
740         fsp = VFSTOPCFS(vp->v_vfsp);
741         error = pc_lockfs(fsp, 0, 0);
742         if (error)
743                 return (error);
744         if ((pcp = VTOPC(vp)) == NULL || pcp->pc_flags & PC_INVAL) {
745                 pc_unlockfs(fsp);
746                 return (EIO);
747         }
748
749         if (fsp->pcfs_flags & PCFS_BOOTPART) {
750                 if (secpolicy_pcfs_modify_bootpartition(cr) != 0) {
751                         pc_unlockfs(fsp);
752                         return (EACCES);
753                 }
754         }
755
756         /*
757          * Change file access modes.
758          * If nobody has write permission, file is marked readonly.
759          * Otherwise file is writable by anyone.
760          */
761         if ((mask & AT_MODE) && (vap->va_mode != (mode_t)-1)) {
762                 if ((vap->va_mode & 0222) == 0)
763                         pcp->pc_entry.pcd_attr |= PCA_RDONLY;
764                 else
765                         pcp->pc_entry.pcd_attr &= ~PCA_RDONLY;
766                 pcp->pc_flags |= PC_CHG;
767         }
768         /*
769          * Truncate file. Must have write permission.
770          */
771         if ((mask & AT_SIZE) && (vap->va_size != (u_offset_t)-1)) {
772                 if (pcp->pc_entry.pcd_attr & PCA_RDONLY) {
773                         error = EACCES;
774                         goto out;
775                 }
776                 if (vap->va_size > UINT32_MAX) {
777                         error = EFBIG;
778                         goto out;
779                 }
780                 error = pc_truncate(pcp, (uint_t)vap->va_size);
781
782                 if (error)
783                         goto out;
784
785                 if (vap->va_size == 0)
786                         vnevent_truncate(vp, ct);
787         }
```

```
788         /*
789          * Change file modified times.
790          */
791         if (mask & (AT_MTIME | AT_CTIME)) {
792                 /*
793                  * If SysV-compatible option to set access and
794                  * modified times if privileged, owner, or write access,
795                  * use current time rather than va_mtime.
796                  *
797                  * XXX - va_mtime.tv_sec == -1 flags this.
798                  */
799                 timep = &vap->va_mtime;
800                 if (vap->va_mtime.tv_sec == -1) {
801                         gethrestime(&now);
802                         timep = &now;
803                 }
804                 if ((fsp->pcfs_flags & PCFS_NOCLAMPTIME) == 0 &&
805                     timep->tv_sec > INT32_MAX) {
806                         error = EOVERFLOW;
807                         goto out;
808                 }
809                 error = pc_tvtopct(timep, &pcp->pc_entry.pcd_mtime);
810                 if (error)
811                         goto out;
812                 pcp->pc_flags |= PC_CHG;
813         }
814         /*
815          * Change file access times.
816          */
817         if (mask & AT_ATIME) {
818                 /*
819                  * If SysV-compatible option to set access and
820                  * modified times if privileged, owner, or write access,
821                  * use current time rather than va_mtime.
822                  *
823                  * XXX - va_atime.tv_sec == -1 flags this.
824                  */
825                 struct pctime   atime;

827                 timep = &vap->va_atime;
828                 if (vap->va_atime.tv_sec == -1) {
829                         gethrestime(&now);
830                         timep = &now;
831                 }
832                 if ((fsp->pcfs_flags & PCFS_NOCLAMPTIME) == 0 &&
833                     timep->tv_sec > INT32_MAX) {
834                         error = EOVERFLOW;
835                         goto out;
836                 }
837                 error = pc_tvtopct(timep, &atime);
838                 if (error)
839                         goto out;
840                 pcp->pc_entry.pcd_ladate = atime.pct_date;
841                 pcp->pc_flags |= PC_CHG;
842         }
843 out:
844         pc_unlockfs(fsp);
845         return (error);
846 }


849 /*ARGSUSED*/
850 static int
851 pcfs_access(
852         struct vnode *vp,
853         int mode,
```



```
854         int flags,
855         struct cred *cr,
856         caller_context_t *ct)
857 {
858         struct pcnode *pcp;
859         struct pcfs *fsp;


862         fsp = VFSTOPCFS(vp->v_vfsp);

864         if ((pcp = VTOPC(vp)) == NULL || pcp->pc_flags & PC_INVAL)
865                 return (EIO);
866         if ((mode & VWRITE) && (pcp->pc_entry.pcd_attr & PCA_RDONLY))
867                 return (EACCES);

869         /*
870          * If this is a boot partition, privileged users have full access while
871          * others have read-only access.
872          */
873         if (fsp->pcfs_flags & PCFS_BOOTPART) {
874                 if ((mode & VWRITE) &&
875                     secpolicy_pcfs_modify_bootpartition(cr) != 0)
876                         return (EACCES);
877         }
878         return (0);
879 }


882 /*ARGSUSED*/
883 static int
884 pcfs_fsync(
885         struct vnode *vp,
886         int syncflag,
887         struct cred *cr,
888         caller_context_t *ct)
889 {
890         struct pcfs *fsp;
891         struct pcnode *pcp;
892         int error;

894         fsp = VFSTOPCFS(vp->v_vfsp);
895         if (error = pc_verify(fsp))
896                 return (error);
897         error = pc_lockfs(fsp, 0, 0);
898         if (error)
899                 return (error);
900         if ((pcp = VTOPC(vp)) == NULL || pcp->pc_flags & PC_INVAL) {
901                 pc_unlockfs(fsp);
902                 return (EIO);
903         }
904         rw_enter(&pcnodes_lock, RW_WRITER);
905         error = pc_nodesync(pcp);
906         rw_exit(&pcnodes_lock);
907         pc_unlockfs(fsp);
908         return (error);
909 }


912 /*ARGSUSED*/
913 static void
914 pcfs_inactive(
915         struct vnode *vp,
916         struct cred *cr,
917         caller_context_t *ct)
918 {
919         struct pcnode *pcp;
```

```
920         struct pcfs *fsp;
921         int error;

923         fsp = VFSTOPCFS(vp->v_vfsp);
924         error = pc_lockfs(fsp, 0, 1);

926         /*
927          * If the filesystem was umounted by force, all dirty
928          * pages associated with this vnode are invalidated
929          * and then the vnode will be freed.
930          */
931         if (vp->v_vfsp->vfs_flag & VFS_UNMOUNTED) {
932                 pcp = VTOPC(vp);
933                 if (vn_has_cached_data(vp)) {
934                         (void) pvn_vplist_dirty(vp, (u_offset_t)0,
935                             pcfs_putapage, B_INVAL, (struct cred *)NULL);
936                 }
937                 remque(pcp);
938                 if (error == 0)
939                         pc_unlockfs(fsp);
940                 vn_free(vp);
941                 kmem_free(pcp, sizeof (struct pcnode));
942                 VFS_RELE(PCFSTOVFS(fsp));
943                 return;
944         }

946         mutex_enter(&vp->v_lock);
947         ASSERT(vp->v_count >= 1);
948         if (vp->v_count > 1) {
949                 vp->v_count--;  /* release our hold from vn_rele */
950                 mutex_exit(&vp->v_lock);
951                 pc_unlockfs(fsp);
952                 return;
953         }
954         mutex_exit(&vp->v_lock);

956         /*
957          * Check again to confirm that no intervening I/O error
958          * with a subsequent pc_diskchanged() call has released
959          * the pcnode. If it has then release the vnode as above.
960          */
961         pcp = VTOPC(vp);
962         if (pcp == NULL || pcp->pc_flags & PC_INVAL) {
963                 if (vn_has_cached_data(vp))
964                         (void) pvn_vplist_dirty(vp, (u_offset_t)0,
965                             pcfs_putapage, B_INVAL | B_TRUNC,
966                             (struct cred *)NULL);
967         }

969         if (pcp == NULL) {
970                 vn_free(vp);
971         } else {
972                 pc_rele(pcp);
973         }

975         if (!error)
976                 pc_unlockfs(fsp);
977 }

979 /*ARGSUSED*/
980 static int
981 pcfs_lookup(
982         struct vnode *dvp,
983         char *nm,
984         struct vnode **vpp,
985         struct pathname *pnp,
```



```
986         int flags,
987         struct vnode *rdir,
988         struct cred *cr,
989         caller_context_t *ct,
990         int *direntflags,
991         pathname_t *realpnp)
992 {
993         struct pcfs *fsp;
994         struct pcnode *pcp;
995         int error;

997         /*
998          * If the filesystem was umounted by force, return immediately.
999          */
1000        if (dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1001                return (EIO);

1003        /*
1004         * verify that the dvp is still valid on the disk
1005         */
1006        fsp = VFSTOPCFS(dvp->v_vfsp);
1007        if (error = pc_verify(fsp))
1008                return (error);
1009        error = pc_lockfs(fsp, 0, 0);
1010        if (error)
1011                return (error);
1012        if (VTOPC(dvp) == NULL || VTOPC(dvp)->pc_flags & PC_INVAL) {
1013                pc_unlockfs(fsp);
1014                return (EIO);
1015        }
1016        /*
1017         * Null component name is a synonym for directory being searched.
1018         */
1019        if (*nm == '\0') {
1020                VN_HOLD(dvp);
1021                *vpp = dvp;
1022                pc_unlockfs(fsp);
1023                return (0);
1024        }

1026        error = pc_dirlook(VTOPC(dvp), nm, &pcp);
1027        if (!error) {
1028                *vpp = PCTOV(pcp);
1029                pcp->pc_flags |= PC_EXTERNAL;
1030        }
1031        pc_unlockfs(fsp);
1032        return (error);
1033 }


1036 /*ARGSUSED*/
1037 static int
1038 pcfs_create(
1039         struct vnode *dvp,
1040         char *nm,
1041         struct vattr *vap,
1042         enum vcexcl exclusive,
1043         int mode,
1044         struct vnode **vpp,
1045         struct cred *cr,
1046         int flag,
1047         caller_context_t *ct,
1048         vsecattr_t *vsecp)
1049 {
1050         int error;
1051         struct pcnode *pcp;
```

```
1052         struct vnode *vp;
1053         struct pcfs *fsp;
1054 
1055         /*
1056          * can't create directories. use pcfs_mkdir.
1057          * can't create anything other than files.
1058          */
1059         if (vap->va_type == VDIR)
1060                 return (EISDIR);
1061         else if (vap->va_type != VREG)
1062                 return (EINVAL);
1063 
1064         pcp = NULL;
1065         fsp = VFSTOPCFS(dvp->v_vfsp);
1066         error = pc_lockfs(fsp, 0, 0);
1067         if (error)
1068                 return (error);
1069         if (VTOPC(dvp) == NULL || VTOPC(dvp)->pc_flags & PC_INVAL) {
1070                 pc_unlockfs(fsp);
1071                 return (EIO);
1072         }
1073 
1074         if (fsp->pcfs_flags & PCFS_BOOTPART) {
1075                 if (secpolicy_pcfs_modify_bootpartition(cr) != 0) {
1076                         pc_unlockfs(fsp);
1077                         return (EACCES);
1078                 }
1079         }
1080 
1081         if (*nm == '\0') {
1082                 /*
1083                  * Null component name refers to the directory itself.
1084                  */
1085                 VN_HOLD(dvp);
1086                 pcp = VTOPC(dvp);
1087                 error = EEXIST;
1088         } else {
1089                 error = pc_direnter(VTOPC(dvp), nm, vap, &pcp);
1090         }
1091         /*
1092          * if file exists and this is a nonexclusive create,
1093          * check for access permissions
1094          */
1095         if (error == EEXIST) {
1096                 vp = PCTOV(pcp);
1097                 if (exclusive == NONEXCL) {
1098                         if (vp->v_type == VDIR) {
1099                                 error = EISDIR;
1100                         } else if (mode) {
1101                                 error = pcfs_access(PCTOV(pcp), mode, 0,
1102                                     cr, ct);
1103                         } else {
1104                                 error = 0;
1105                         }
1106                 }
1107                 if (error) {
1108                         VN_RELE(PCTOV(pcp));
1109                 } else if ((vp->v_type == VREG) && (vap->va_mask & AT_SIZE) &&
1110                     (vap->va_size == 0)) {
1111                         error = pc_truncate(pcp, 0L);
1112                         if (error) {
1113                                 VN_RELE(PCTOV(pcp));
1114                         } else {
1115                                 vnevent_create(PCTOV(pcp), ct);
1116                         }
1117                 }
```



```
1118         }
1119         if (error) {
1120                 pc_unlockfs(fsp);
1121                 return (error);
1122         }
1123         *vpp = PCTOV(pcp);
1124         pcp->pc_flags |= PC_EXTERNAL;
1125         pc_unlockfs(fsp);
1126         return (error);
1127 }
1128 
1129 /*ARGSUSED*/
1130 static int
1131 pcfs_remove(
1132         struct vnode *vp,
1133         char *nm,
1134         struct cred *cr,
1135         caller_context_t *ct,
1136         int flags)
1137 {
1138         struct pcfs *fsp;
1139         struct pcnode *pcp;
1140         int error;
1141 
1142         fsp = VFSTOPCFS(vp->v_vfsp);
1143         if (error = pc_verify(fsp))
1144                 return (error);
1145         error = pc_lockfs(fsp, 0, 0);
1146         if (error)
1147                 return (error);
1148         if ((pcp = VTOPC(vp)) == NULL || pcp->pc_flags & PC_INVAL) {
1149                 pc_unlockfs(fsp);
1150                 return (EIO);
1151         }
1152         if (fsp->pcfs_flags & PCFS_BOOTPART) {
1153                 if (secpolicy_pcfs_modify_bootpartition(cr) != 0) {
1154                         pc_unlockfs(fsp);
1155                         return (EACCES);
1156                 }
1157         }
1158         error = pc_dirremove(pcp, nm, (struct vnode *)0, VREG, ct);
1159         pc_unlockfs(fsp);
1160         return (error);
1161 }
1162 
1163 /*
1164  * Rename a file or directory
1165  * This rename is restricted to only rename files within a directory.
1166  * XX should make rename more general
1167  */
1168 /*ARGSUSED*/
1169 static int
1170 pcfs_rename(
1171         struct vnode *sdvp,             /* old (source) parent vnode */
1172         char *snm,                      /* old (source) entry name */
1173         struct vnode *tdvp,             /* new (target) parent vnode */
1174         char *tnm,                      /* new (target) entry name */
1175         struct cred *cr,
1176         caller_context_t *ct,
1177         int flags)
1178 {
1179         struct pcfs *fsp;
1180         struct pcnode *dp;      /* parent pcnode */
1181         struct pcnode *tdp;
1182         int error;
```

```
1184         fsp = VFSTOPCFS(sdvp->v_vfsp);
1185         if (error = pc_verify(fsp))
1186                 return (error);

1188         /*
1189          * make sure we can muck with this directory.
1190          */
1191         error = pcfs_access(sdvp, VWRITE, 0, cr, ct);
1192         if (error) {
1193                 return (error);
1194         }
1195         error = pc_lockfs(fsp, 0, 0);
1196         if (error)
1197                 return (error);
1198         if (((dp = VTOPC(sdvp)) == NULL) || ((tdp = VTOPC(tdvp)) == NULL) ||
1199             (dp->pc_flags & PC_INVAL) || (tdp->pc_flags & PC_INVAL)) {
1200                 pc_unlockfs(fsp);
1201                 return (EIO);
1202         }
1203         error = pc_rename(dp, tdp, snm, tnm, ct);
1204         pc_unlockfs(fsp);
1205         return (error);
1206 }

1208 /*ARGSUSED*/
1209 static int
1210 pcfs_mkdir(
1211         struct vnode *dvp,
1212         char *nm,
1213         struct vattr *vap,
1214         struct vnode **vpp,
1215         struct cred *cr,
1216         caller_context_t *ct,
1217         int flags,
1218         vsecattr_t *vsecp)
1219 {
1220         struct pcfs *fsp;
1221         struct pcnode *pcp;
1222         int error;

1224         fsp = VFSTOPCFS(dvp->v_vfsp);
1225         if (error = pc_verify(fsp))
1226                 return (error);
1227         error = pc_lockfs(fsp, 0, 0);
1228         if (error)
1229                 return (error);
1230         if (VTOPC(dvp) == NULL || VTOPC(dvp)->pc_flags & PC_INVAL) {
1231                 pc_unlockfs(fsp);
1232                 return (EIO);
1233         }

1235         if (fsp->pcfs_flags & PCFS_BOOTPART) {
1236                 if (secpolicy_pcfs_modify_bootpartition(cr) != 0) {
1237                         pc_unlockfs(fsp);
1238                         return (EACCES);
1239                 }
1240         }

1242         error = pc_direnter(VTOPC(dvp), nm, vap, &pcp);

1244         if (!error) {
1245                 pcp -> pc_flags |= PC_EXTERNAL;
1246                 *vpp = PCTOV(pcp);
1247         } else if (error == EEXIST) {
1248                 VN_RELE(PCTOV(pcp));
1249         }
```



```
1250         pc_unlockfs(fsp);
1251         return (error);
1252 }

1254 /*ARGSUSED*/
1255 static int
1256 pcfs_rmdir(
1257         struct vnode *dvp,
1258         char *nm,
1259         struct vnode *cdir,
1260         struct cred *cr,
1261         caller_context_t *ct,
1262         int flags)
1263 {
1264         struct pcfs *fsp;
1265         struct pcnode *pcp;
1266         int error;

1268         fsp = VFSTOPCFS(dvp -> v_vfsp);
1269         if (error = pc_verify(fsp))
1270                 return (error);
1271         if (error = pc_lockfs(fsp, 0, 0))
1272                 return (error);

1274         if ((pcp = VTOPC(dvp)) == NULL || pcp->pc_flags & PC_INVAL) {
1275                 pc_unlockfs(fsp);
1276                 return (EIO);
1277         }

1279         if (fsp->pcfs_flags & PCFS_BOOTPART) {
1280                 if (secpolicy_pcfs_modify_bootpartition(cr) != 0) {
1281                         pc_unlockfs(fsp);
1282                         return (EACCES);
1283                 }
1284         }

1286         error = pc_dirremove(pcp, nm, cdir, VDIR, ct);
1287         pc_unlockfs(fsp);
1288         return (error);
1289 }

1291 /*
1292  * read entries in a directory.
1293  * we must convert pc format to unix format
1294  */

1296 /*ARGSUSED*/
1297 static int
1298 pcfs_readdir(
1299         struct vnode *dvp,
1300         struct uio *uiop,
1301         struct cred *cr,
1302         int *eofp,
1303         caller_context_t *ct,
1304         int flags)
1305 {
1306         struct pcnode *pcp;
1307         struct pcfs *fsp;
1308         struct pcdir *ep;
1309         struct buf *bp = NULL;
1310         offset_t offset;
1311         int boff;
1312         struct pc_dirent lbp;
1313         struct pc_dirent *ld = &lbp;
1314         int error;
```

```
1316         /*
1317          * If the filesystem was umounted by force, return immediately.
1318          */
1319         if (dvp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1320                 return (EIO);

1322         if ((uiop->uio_iovcnt != 1) ||
1323             (uiop->uio_loffset % sizeof (struct pcdir)) != 0) {
1324                 return (EINVAL);
1325         }
1326         fsp = VFSTOPCFS(dvp->v_vfsp);
1327         /*
1328          * verify that the dp is still valid on the disk
1329          */
1330         if (error = pc_verify(fsp)) {
1331                 return (error);
1332         }
1333         error = pc_lockfs(fsp, 0, 0);
1334         if (error)
1335                 return (error);
1336         if ((pcp = VTOPC(dvp)) == NULL || pcp->pc_flags & PC_INVAL) {
1337                 pc_unlockfs(fsp);
1338                 return (EIO);
1339         }

1341         bzero(ld, sizeof (*ld));

1343         if (eofp != NULL)
1344                 *eofp = 0;
1345         offset = uiop->uio_loffset;

1347         if (dvp->v_flag & VROOT) {
1348                 /*
1349                  * kludge up entries for "." and ".." in the root.
1350                  */
1351                 if (offset == 0) {
1352                         (void) strcpy(ld->d_name, ".");
1353                         ld->d_reclen = DIRENT64_RECLEN(1);
1354                         ld->d_off = (off64_t)sizeof (struct pcdir);
1355                         ld->d_ino = (ino64_t)UINT_MAX;
1356                         if (ld->d_reclen > uiop->uio_resid) {
1357                                 pc_unlockfs(fsp);
1358                                 return (ENOSPC);
1359                         }
1360                         (void) uiomove(ld, ld->d_reclen, UIO_READ, uiop);
1361                         uiop->uio_loffset = ld->d_off;
1362                         offset = uiop->uio_loffset;
1363                 }
1364                 if (offset == sizeof (struct pcdir)) {
1365                         (void) strcpy(ld->d_name, "..");
1366                         ld->d_reclen = DIRENT64_RECLEN(2);
1367                         if (ld->d_reclen > uiop->uio_resid) {
1368                                 pc_unlockfs(fsp);
1369                                 return (ENOSPC);
1370                         }
1371                         ld->d_off = (off64_t)(uiop->uio_loffset +
1372                             sizeof (struct pcdir));
1373                         ld->d_ino = (ino64_t)UINT_MAX;
1374                         (void) uiomove(ld, ld->d_reclen, UIO_READ, uiop);
1375                         uiop->uio_loffset = ld->d_off;
1376                         offset = uiop->uio_loffset;
1377                 }
1378                 offset -= 2 * sizeof (struct pcdir);
1379                 /* offset now has the real offset value into directory file */
1380         }
```



```
1382         for (;;) {
1383                 boff = pc_blkoff(fsp, offset);
1384                 if (boff == 0 || bp == NULL || boff >= bp->b_bcount) {
1385                         if (bp != NULL) {
1386                                 brelse(bp);
1387                                 bp = NULL;
1388                         }
1389                         error = pc_blkatoff(pcp, offset, &bp, &ep);
1390                         if (error) {
1391                                 if (error == ENOENT) {
1392                                         error = 0;
1393                                         if (eofp)
1394                                                 *eofp = 1;
1395                                 }
1396                                 break;
1397                         }
1398                 }
1399                 if (ep->pcd_filename[0] == PCD_UNUSED) {
1400                         if (eofp)
1401                                 *eofp = 1;
1402                         break;
1403                 }
1404                 /*
1405                  * Don't display label because it may contain funny characters.
1406                  */
1407                 if (ep->pcd_filename[0] == PCD_ERASED) {
1408                         uiop->uio_loffset += sizeof (struct pcdir);
1409                         offset += sizeof (struct pcdir);
1410                         ep++;
1411                         continue;
1412                 }
1413                 if (PCDL_IS_LFN(ep)) {
1414                         if (pc_read_long_fn(dvp, uiop, ld, &ep, &offset, &bp) !=
1415                             0)
1416                                 break;
1417                         continue;
1418                 }

1420                 if (pc_read_short_fn(dvp, uiop, ld, &ep, &offset, &bp) != 0)
1421                         break;
1422         }
1423         if (bp)
1424                 brelse(bp);
1425         pc_unlockfs(fsp);
1426         return (error);
1427 }


1430 /*
1431  * Called from pvn_getpages to get a particular page.  When we are called
1432  * the pcfs is already locked.
  29  * Called from pvn_getpages or pcfs_getpage to get a particular page.
  30  * When we are called the pcfs is already locked.
1433  */
1434 /*ARGSUSED*/
1435 static int
1436 pcfs_getapage(
1437         struct vnode *vp,
1438         u_offset_t off,
1439         size_t len,
1440         uint_t *protp,
1441         page_t *pl[],           /* NULL if async IO is requested */
1442         size_t plsz,
1443         struct seg *seg,
1444         caddr_t addr,
1445         enum seg_rw rw,
```

```
1446         struct cred *cr)
1447 {
1448         struct pcnode *pcp;
1449         struct pcfs *fsp = VFSTOPCFS(vp->v_vfsp);
1450         struct vnode *devvp;
1451         page_t *pp;
1452         page_t *pagefound;
1453         int err;

1455         /*
1456          * If the filesystem was umounted by force, return immediately.
1457          */
1458         if (vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)
1459                 return (EIO);

1461         PC_DPRINTF3(5, "pcfs_getapage: vp=%p off=%lld len=%lu\n",
1462             (void *)vp, off, len);

1464         if ((pcp = VTOPC(vp)) == NULL || pcp->pc_flags & PC_INVAL)
1465                 return (EIO);
1466         devvp = fsp->pcfs_devvp;

1468         /* pcfs doesn't do readaheads */
1469         if (pl == NULL)
1470                 return (0);

1472         pl[0] = NULL;
1473         err = 0;
1474         /*
1475          * If the accessed time on the pcnode has not already been
1476          * set elsewhere (e.g. for read/setattr) we set the time now.
1477          * This gives us approximate modified times for mmap'ed files
1478          * which are accessed via loads in the user address space.
1479          */
1480         if ((pcp->pc_flags & PC_ACC) == 0 &&
1481             ((fsp->pcfs_vfs->vfs_flag & VFS_RDONLY) == 0)) {
1482                 pc_mark_acc(fsp, pcp);
1483         }
1484 reread:
1485         if ((pagefound = page_exists(vp, off)) == NULL) {
1486                 /*
1487                  * Need to really do disk IO to get the page(s).
1488                  */
1489                 struct buf *bp;
1490                 daddr_t lbn, bn;
1491                 u_offset_t io_off;
1492                 size_t io_len;
1493                 u_offset_t lbnoff, xferoffset;
1494                 u_offset_t pgoff;
1495                 uint_t  xfersize;
1496                 int err1;

1498                 lbn = pc_lblkno(fsp, off);
1499                 lbnoff = off & ~(fsp->pcfs_clsize - 1);
1500                 xferoffset = off & ~(fsp->pcfs_secsize - 1);

1502                 pp = pvn_read_kluster(vp, off, seg, addr, &io_off, &io_len,
1503                     off, (size_t)MIN(pc_blksize(fsp, pcp, off), PAGESIZE), 0);
1504                 if (pp == NULL)
1505                         /*
1506                          * XXX - If pcfs is made MT-hot, this should go
1507                          * back to reread.
1508                          */
1509                         panic("pcfs_getapage pvn_read_kluster");

1511                 for (pgoff = 0; pgoff < PAGESIZE && xferoffset < pcp->pc_size;
```



```
1512                     pgoff += xfersize,
1513                     lbn +=  howmany(xfersize, fsp->pcfs_clsize),
1514                     lbnoff += xfersize, xferoffset += xfersize) {
1515                         /*
1516                          * read as many contiguous blocks as possible to
1517                          * fill this page
1518                          */
1519                         xfersize = PAGESIZE - pgoff;
1520                         err1 = pc_bmap(pcp, lbn, &bn, &xfersize);
1521                         if (err1) {
1522                                 PC_DPRINTF1(1, "pc_getapage err=%d", err1);
1523                                 err = err1;
1524                                 goto out;
1525                         }
1526                         bp = pageio_setup(pp, xfersize, devvp, B_READ);
1527                         bp->b_edev = devvp->v_rdev;
1528                         bp->b_dev = cmpdev(devvp->v_rdev);
1529                         bp->b_blkno = bn + btodt(xferoffset - lbnoff);
1530                         bp->b_un.b_addr = (caddr_t)(uintptr_t)pgoff;
1531                         bp->b_file = vp;
1532                         bp->b_offset = (offset_t)(off + pgoff);

1534                         (void) bdev_strategy(bp);

1536                         lwp_stat_update(LWP_STAT_INBLK, 1);

1538                         if (err == 0)
1539                                 err = biowait(bp);
1540                         else
1541                                 (void) biowait(bp);
1542                         pageio_done(bp);
1543                         if (err)
1544                                 goto out;
1545                 }
1546                 if (pgoff < PAGESIZE) {
1547                         pagezero(pp->p_prev, pgoff, PAGESIZE - pgoff);
1548                 }
1549                 pvn_plist_init(pp, pl, plsz, off, io_len, rw);
1550         }
1551 out:
1552         if (err) {
1553                 if (pp != NULL)
1554                         pvn_read_done(pp, B_ERROR);
1555                 return (err);
1556         }

1558         if (pagefound) {
1559                 /*
1560                  * Page exists in the cache, acquire the "shared"
1561                  * lock.  If this fails, go back to reread.
1562                  */
1563                 if ((pp = page_lookup(vp, off, SE_SHARED)) == NULL) {
1564                         goto reread;
1565                 }
1566                 pl[0] = pp;
1567                 pl[1] = NULL;
1568         }
1569         return (err);
1570 }

1572 /*
1573  * Return all the pages from [off..off+len] in given file
1574  */
1575 /* ARGSUSED */
1576 static int
1577 pcfs_getpage(
```

```
1578         struct vnode *vp,
1579         offset_t off,
1580         size_t len,
1581         uint_t *protp,
1582         page_t *pl[],
1583         size_t plsz,
1584         struct seg *seg,
1585         caddr_t addr,
1586         enum seg_rw rw,
1587         struct cred *cr,
1588         caller_context_t *ct)
1589 {
1590         struct pcfs *fsp = VFSTOPCFS(vp->v_vfsp);
1591         int err;

1593         PC_DPRINTF0(6, "pcfs_getpage\n");
1594         if (err = pc_verify(fsp))
1595                 return (err);
1596         if (vp->v_flag & VNOMAP)
1597                 return (ENOSYS);
1598         ASSERT(off <= UINT32_MAX);
1599         err = pc_lockfs(fsp, 0, 0);
1600         if (err)
1601                 return (err);
1602         if (protp != NULL)
1603                 *protp = PROT_ALL;

1605         ASSERT((off & PAGEOFFSET) == 0);
1606         err = pvn_getpages(pcfs_getapage, vp, off, len, protp, pl, plsz,
1607             seg, addr, rw, cr);

 204         if (len <= PAGESIZE) {
 205                 err = pcfs_getapage(vp, off, len, protp, pl,
 206                     plsz, seg, addr, rw, cr);
 207         } else {
 208                 err = pvn_getpages(pcfs_getapage, vp, off,
 209                     len, protp, pl, plsz, seg, addr, rw, cr);
 210         }
1609         pc_unlockfs(fsp);
1610         return (err);
1611 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   66491 Thu Jan  8 09:14:36 2015
new/usr/src/uts/common/fs/specfs/specvnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

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

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

#include <sys/types.h>
#include <sys/thread.h>
#include <sys/t_lock.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/bitmap.h>
#include <sys/buf.h>
#include <sys/cmn_err.h>
#include <sys/conf.h>
#include <sys/ddi.h>
#include <sys/debug.h>
#include <sys/dkio.h>
#include <sys/errno.h>
#include <sys/time.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/file.h>
#include <sys/kmem.h>
#include <sys/mman.h>
#include <sys/open.h>
#include <sys/swap.h>
```



```
#include <sys/sysmacros.h>
#include <sys/uio.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/vnode.h>
#include <sys/stat.h>
#include <sys/poll.h>
#include <sys/stream.h>
#include <sys/strsubr.h>
#include <sys/policy.h>
#include <sys/devpolicy.h>

#include <sys/proc.h>
#include <sys/user.h>
#include <sys/session.h>
#include <sys/vmsystm.h>
#include <sys/vtrace.h>
#include <sys/pathname.h>

#include <sys/fs/snode.h>

#include <vm/seg.h>
#include <vm/seg_map.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <vm/seg_dev.h>
#include <vm/seg_vn.h>

#include <fs/fs_subr.h>

#include <sys/esunddi.h>
#include <sys/autoconf.h>
#include <sys/sunndi.h>
#include <sys/contract/device_impl.h>


static int spec_open(struct vnode **, int, struct cred *, caller_context_t *);
static int spec_close(struct vnode *, int, int, offset_t, struct cred *,
        caller_context_t *);
static int spec_read(struct vnode *, struct uio *, int, struct cred *,
        caller_context_t *);
static int spec_write(struct vnode *, struct uio *, int, struct cred *,
        caller_context_t *);
static int spec_ioctl(struct vnode *, int, intptr_t, int, struct cred *, int *,
        caller_context_t *);
static int spec_getattr(struct vnode *, struct vattr *, int, struct cred *,
        caller_context_t *);
static int spec_setattr(struct vnode *, struct vattr *, int, struct cred *,
        caller_context_t *);
static int spec_access(struct vnode *, int, int, struct cred *,
        caller_context_t *);
static int spec_create(struct vnode *, char *, vattr_t *, enum vcexcl, int,
        struct vnode **, struct cred *, int, caller_context_t *, vsecattr_t *);
static int spec_fsync(struct vnode *, int, struct cred *, caller_context_t *);
static void spec_inactive(struct vnode *, struct cred *, caller_context_t *);
static int spec_fid(struct vnode *, struct fid *, caller_context_t *);
static int spec_seek(struct vnode *, offset_t, offset_t *, caller_context_t *);
static int spec_frlock(struct vnode *, int, struct flock64 *, int, offset_t,
        struct flk_callback *, struct cred *, caller_context_t *);
static int spec_realvp(struct vnode *, struct vnode **, caller_context_t *);

static int spec_getpage(struct vnode *, offset_t, size_t, uint_t *, page_t **,
        size_t, struct seg *, caddr_t, enum seg_rw, struct cred *,
        caller_context_t *);
static int spec_putapage(struct vnode *, page_t *, u_offset_t *, size_t *, int,
        struct cred *);
```

```
128 static struct buf *spec_startio(struct vnode *, page_t *, u_offset_t, size_t,
129         int);
130 static int spec_getapage(struct vnode *, u_offset_t, size_t, uint_t *,
131         page_t **, size_t, struct seg *, caddr_t, enum seg_rw, struct cred *);
132 static int spec_map(struct vnode *, offset_t, struct as *, caddr_t *, size_t,
133         uchar_t, uchar_t, uint_t, struct cred *, caller_context_t *);
134 static int spec_addmap(struct vnode *, offset_t, struct as *, caddr_t, size_t,
135         uchar_t, uchar_t, uint_t, struct cred *, caller_context_t *);
136 static int spec_delmap(struct vnode *, offset_t, struct as *, caddr_t, size_t,
137         uint_t, uint_t, uint_t, struct cred *, caller_context_t *);

139 static int spec_poll(struct vnode *, short, int, short *, struct pollhead **,
140         caller_context_t *);
141 static int spec_dump(struct vnode *, caddr_t, offset_t, offset_t,
142     caller_context_t *);
143 static int spec_pageio(struct vnode *, page_t *, u_offset_t, size_t, int,
144     cred_t *, caller_context_t *);

146 static int spec_getsecattr(struct vnode *, vsecattr_t *, int, struct cred *,
147         caller_context_t *);
148 static int spec_setsecattr(struct vnode *, vsecattr_t *, int, struct cred *,
149         caller_context_t *);
150 static int spec_pathconf(struct vnode *, int, ulong_t *, struct cred *,
151         caller_context_t *);

153 #define SN_HOLD(csp)    { \
154         mutex_enter(&csp->s_lock); \
155         csp->s_count++; \
156         mutex_exit(&csp->s_lock); \
157 }

159 #define SN_RELE(csp)    { \
160         mutex_enter(&csp->s_lock); \
161         csp->s_count--; \
162         ASSERT((csp->s_count > 0) || (csp->s_vnode->v_stream == NULL)); \
163         mutex_exit(&csp->s_lock); \
164 }

166 #define S_ISFENCED(sp)  ((VTOS((sp)->s_commonvp))->s_flag & SFENCED)

168 struct vnodeops *spec_vnodeops;

170 /*
171  * *PLEASE NOTE*: If you add new entry points to specfs, do
172  * not forget to add support for fencing. A fenced snode
173  * is indicated by the SFENCED flag in the common snode.
174  * If a snode is fenced, determine if your entry point is
175  * a configuration operation (Example: open), a detection
176  * operation (Example: gettattr), an I/O operation (Example: ioctl())
177  * or an unconfiguration operation (Example: close). If it is
178  * a configuration or detection operation, fail the operation
179  * for a fenced snode with an ENXIO or EIO as appropriate. If
180  * it is any other operation, let it through.
181  */

183 const fs_operation_def_t spec_vnodeops_template[] = {
184         VOPNAME_OPEN,           { .vop_open = spec_open },
185         VOPNAME_CLOSE,          { .vop_close = spec_close },
186         VOPNAME_READ,           { .vop_read = spec_read },
187         VOPNAME_WRITE,          { .vop_write = spec_write },
188         VOPNAME_IOCTL,          { .vop_ioctl = spec_ioctl },
189         VOPNAME_GETATTR,        { .vop_getattr = spec_getattr },
190         VOPNAME_SETATTR,        { .vop_setattr = spec_setattr },
191         VOPNAME_ACCESS,         { .vop_access = spec_access },
192         VOPNAME_CREATE,         { .vop_create = spec_create },
193         VOPNAME_FSYNC,          { .vop_fsync = spec_fsync },
```



```
194         VOPNAME_INACTIVE,       { .vop_inactive = spec_inactive },
195         VOPNAME_FID,            { .vop_fid = spec_fid },
196         VOPNAME_SEEK,           { .vop_seek = spec_seek },
197         VOPNAME_PATHCONF,       { .vop_pathconf = spec_pathconf },
198         VOPNAME_FRLOCK,         { .vop_frlock = spec_frlock },
199         VOPNAME_REALVP,         { .vop_realvp = spec_realvp },
200         VOPNAME_GETPAGE,        { .vop_getpage = spec_getpage },
201         VOPNAME_PUTPAGE,        { .vop_putpage = spec_putpage },
202         VOPNAME_MAP,            { .vop_map = spec_map },
203         VOPNAME_ADDMAP,         { .vop_addmap = spec_addmap },
204         VOPNAME_DELMAP,         { .vop_delmap = spec_delmap },
205         VOPNAME_POLL,           { .vop_poll = spec_poll },
206         VOPNAME_DUMP,           { .vop_dump = spec_dump },
207         VOPNAME_PAGEIO,         { .vop_pageio = spec_pageio },
208         VOPNAME_SETSECATTR,     { .vop_setsecattr = spec_setsecattr },
209         VOPNAME_GETSECATTR,     { .vop_getsecattr = spec_getsecattr },
210         NULL,                   NULL
211 };

213 /*
214  * Return address of spec_vnodeops
215  */
216 struct vnodeops *
217 spec_getvnodeops(void)
218 {
219         return (spec_vnodeops);
220 }

222 extern vnode_t *rconsvp;

224 /*
225  * Acquire the serial lock on the common snode.
226  */
227 #define LOCK_CSP(csp)                   (void) spec_lockcsp(csp, 0, 1, 0)
228 #define LOCKHOLD_CSP_SIG(csp)           spec_lockcsp(csp, 1, 1, 1)
229 #define SYNCHOLD_CSP_SIG(csp, intr)     spec_lockcsp(csp, intr, 0, 1)

231 typedef enum {
232         LOOP,
233         INTR,
234         SUCCESS
235 } slock_ret_t;

237 /*
238  * Synchronize with active SLOCKED snode, optionally checking for a signal and
239  * optionally returning with SLOCKED set and SN_HOLD done.  The 'intr'
240  * argument determines if the thread is interruptible by a signal while
241  * waiting, the function returns INTR if interrupted while there is another
242  * thread closing this snonde and LOOP if interrupted otherwise.
243  * When SUCCESS is returned the 'hold' argument determines if the open
244  * count (SN_HOLD) has been incremented and the 'setlock' argument
245  * determines if the function returns with SLOCKED set.
246  */
247 static slock_ret_t
248 spec_lockcsp(struct snode *csp, int intr, int setlock, int hold)
249 {
250         slock_ret_t ret = SUCCESS;
251         mutex_enter(&csp->s_lock);
252         while (csp->s_flag & SLOCKED) {
253                 csp->s_flag |= SWANT;
254                 if (intr) {
255                         if (!cv_wait_sig(&csp->s_cv, &csp->s_lock)) {
256                                 if (csp->s_flag & SCLOSING)
257                                         ret = INTR;
258                                 else
259                                         ret = LOOP;
```

```
260                                 mutex_exit(&csp->s_lock);
261                                 return (ret);           /* interrupted */
262                         }
263                 } else {
264                         cv_wait(&csp->s_cv, &csp->s_lock);
265                 }
266         }
267         if (setlock)
268                 csp->s_flag |= SLOCKED;
269         if (hold)
270                 csp->s_count++;         /* one more open reference : SN_HOLD */
271         mutex_exit(&csp->s_lock);
272         return (ret);                   /* serialized/locked */
273 }

275 /*
276  * Unlock the serial lock on the common snode
277  */
278 #define UNLOCK_CSP_LOCK_HELD(csp)                       \
279         ASSERT(mutex_owned(&csp->s_lock));              \
280         if (csp->s_flag & SWANT)                        \
281                 cv_broadcast(&csp->s_cv);               \
282         csp->s_flag &= ~(SWANT|SLOCKED);

284 #define UNLOCK_CSP(csp)                                 \
285         mutex_enter(&csp->s_lock);                      \
286         UNLOCK_CSP_LOCK_HELD(csp);                      \
287         mutex_exit(&csp->s_lock);

289 /*
290  * compute/return the size of the device
291  */
292 #define SPEC_SIZE(csp)  \
293         (((csp)->s_flag & SSIZEVALID) ? (csp)->s_size : spec_size(csp))

295 /*
296  * Compute and return the size.  If the size in the common snode is valid then
297  * return it.  If not valid then get the size from the driver and set size in
298  * the common snode.  If the device has not been attached then we don't ask for
299  * an update from the driver- for non-streams SSIZEVALID stays unset until the
300  * device is attached. A stat of a mknod outside /devices (non-devfs) may
301  * report UNKNOWN_SIZE because the device may not be attached yet (SDIPSET not
302  * established in mknod until open time). An stat in /devices will report the
303  * size correctly.  Specfs should always call SPEC_SIZE instead of referring
304  * directly to s_size to initialize/retrieve the size of a device.
305  *
306  * XXX There is an inconsistency between block and raw - "unknown" is
307  * UNKNOWN_SIZE for VBLK and 0 for VCHR(raw).
308  */
309 static u_offset_t
310 spec_size(struct snode *csp)
311 {
312         struct vnode    *cvp = STOV(csp);
313         u_offset_t      size;
314         int             plen;
315         uint32_t        size32;
316         dev_t           dev;
317         dev_info_t      *devi;
318         major_t         maj;
319         uint_t          blksize;
320         int             blkshift;

322         ASSERT((csp)->s_commonvp == cvp);       /* must be common node */

324         /* return cached value */
325         mutex_enter(&csp->s_lock);
```



```
326         if (csp->s_flag & SSIZEVALID) {
327                 mutex_exit(&csp->s_lock);
328                 return (csp->s_size);
329         }

331         /* VOP_GETATTR of mknod has not had devcnt restriction applied */
332         dev = cvp->v_rdev;
333         maj = getmajor(dev);
334         if (maj >= devcnt) {
335                 /* return non-cached UNKNOWN_SIZE */
336                 mutex_exit(&csp->s_lock);
337                 return ((cvp->v_type == VCHR) ? 0 : UNKNOWN_SIZE);
338         }

340         /* establish cached zero size for streams */
341         if (STREAMSTAB(maj)) {
342                 csp->s_size = 0;
343                 csp->s_flag |= SSIZEVALID;
344                 mutex_exit(&csp->s_lock);
345                 return (0);
346         }

348         /*
349          * Return non-cached UNKNOWN_SIZE if not open.
350          *
351          * NB: This check is bogus, calling prop_op(9E) should be gated by
352          * attach, not open. Not having this check however opens up a new
353          * context under which a driver's prop_op(9E) could be called. Calling
354          * prop_op(9E) in this new context has been shown to expose latent
355          * driver bugs (insufficient NULL pointer checks that lead to panic).
356          * We are keeping this open check for now to avoid these panics.
357          */
358         if (csp->s_count == 0) {
359                 mutex_exit(&csp->s_lock);
360                 return ((cvp->v_type == VCHR) ? 0 : UNKNOWN_SIZE);
361         }

363         /* Return non-cached UNKNOWN_SIZE if not attached. */
364         if (((csp->s_flag & SDIPSET) == 0) || (csp->s_dip == NULL) ||
365             !i_ddi_devi_attached(csp->s_dip)) {
366                 mutex_exit(&csp->s_lock);
367                 return ((cvp->v_type == VCHR) ? 0 : UNKNOWN_SIZE);
368         }

370         devi = csp->s_dip;

372         /*
373          * Established cached size obtained from the attached driver. Since we
374          * know the devinfo node, for efficiency we use cdev_prop_op directly
375          * instead of [cb]dev_[Ss]size.
376          */
377         if (cvp->v_type == VCHR) {
378                 size = 0;
379                 plen = sizeof (size);
380                 if (cdev_prop_op(dev, devi, PROP_LEN_AND_VAL_BUF,
381                     DDI_PROP_NOTPROM | DDI_PROP_DONTPASS |
382                     DDI_PROP_CONSUMER_TYPED, "Size", (caddr_t)&size,
383                     &plen) != DDI_PROP_SUCCESS) {
384                         plen = sizeof (size32);
385                         if (cdev_prop_op(dev, devi, PROP_LEN_AND_VAL_BUF,
386                             DDI_PROP_NOTPROM | DDI_PROP_DONTPASS,
387                             "size", (caddr_t)&size32, &plen) ==
388                             DDI_PROP_SUCCESS)
389                                 size = size32;
390                 }
391         } else {
```

```
392                 size = UNKNOWN_SIZE;
393                 plen = sizeof (size);
394                 if (cdev_prop_op(dev, devi, PROP_LEN_AND_VAL_BUF,
395                     DDI_PROP_NOTPROM | DDI_PROP_DONTPASS |
396                     DDI_PROP_CONSUMER_TYPED, "Nblocks", (caddr_t)&size,
397                     &plen) != DDI_PROP_SUCCESS) {
398                         plen = sizeof (size32);
399                         if (cdev_prop_op(dev, devi, PROP_LEN_AND_VAL_BUF,
400                             DDI_PROP_NOTPROM | DDI_PROP_DONTPASS,
401                             "nblocks", (caddr_t)&size32, &plen) ==
402                             DDI_PROP_SUCCESS)
403                                 size = size32;
404                 }
405
406                 if (size != UNKNOWN_SIZE) {
407                         blksize = DEV_BSIZE;            /* default */
408                         plen = sizeof (blksize);
409
410                         /* try to get dev_t specific "blksize" */
411                         if (cdev_prop_op(dev, devi, PROP_LEN_AND_VAL_BUF,
412                             DDI_PROP_NOTPROM | DDI_PROP_DONTPASS,
413                             "blksize", (caddr_t)&blksize, &plen) !=
414                             DDI_PROP_SUCCESS) {
415                                 /*
416                                  * Try for dev_info node "device-blksize".
417                                  * If this fails then blksize will still be
418                                  * DEV_BSIZE default value.
419                                  */
420                                 (void) cdev_prop_op(DDI_DEV_T_ANY, devi,
421                                     PROP_LEN_AND_VAL_BUF,
422                                     DDI_PROP_NOTPROM | DDI_PROP_DONTPASS,
423                                     "device-blksize", (caddr_t)&blksize, &plen);
424                         }
425
426                         /* blksize must be a power of two */
427                         ASSERT(BIT_ONLYONESET(blksize));
428                         blkshift = highbit(blksize) - 1;
429
430                         /* convert from block size to byte size */
431                         if (size < (MAXOFFSET_T >> blkshift))
432                                 size = size << blkshift;
433                         else
434                                 size = UNKNOWN_SIZE;
435                 }
436         }
437
438         csp->s_size = size;
439         csp->s_flag |= SSIZEVALID;
440
441         mutex_exit(&csp->s_lock);
442         return (size);
443 }
444
445 /*
446  * This function deal with vnode substitution in the case of
447  * device cloning.
448  */
449 static int
450 spec_clone(struct vnode **vpp, dev_t newdev, int vtype, struct stdata *stp)
451 {
452         dev_t           dev = (*vpp)->v_rdev;
453         major_t         maj = getmajor(dev);
454         major_t         newmaj = getmajor(newdev);
455         int             sysclone = (maj == clone_major);
456         int             qassociate_used = 0;
457         struct snode    *oldsp, *oldcsp;
```



```
458         struct snode    *newsp, *newcsp;
459         struct vnode    *newvp, *newcvp;
460         dev_info_t      *dip;
461         queue_t         *dq;
462
463         ASSERT(dev != newdev);
464
465         /*
466          * Check for cloning across different drivers.
467          * We only support this under the system provided clone driver
468          */
469         if ((maj != newmaj) && !sysclone) {
470                 cmn_err(CE_NOTE,
471                     "unsupported clone open maj = %u, newmaj = %u",
472                     maj, newmaj);
473                 return (ENXIO);
474         }
475
476         /* old */
477         oldsp = VTOS(*vpp);
478         oldcsp = VTOS(oldsp->s_commonvp);
479
480         /* new */
481         newvp = makespecvp(newdev, vtype);
482         ASSERT(newvp != NULL);
483         newsp = VTOS(newvp);
484         newcvp = newsp->s_commonvp;
485         newcsp = VTOS(newcvp);
486
487         /*
488          * Clones inherit fsid, realvp, and dip.
489          * XXX realvp inherit is not occurring, does fstat of clone work?
490          */
491         newsp->s_fsid = oldsp->s_fsid;
492         if (sysclone) {
493                 newsp->s_flag |= SCLONE;
494                 dip = NULL;
495         } else {
496                 newsp->s_flag |= SSELFCLONE;
497                 dip = oldcsp->s_dip;
498         }
499
500         /*
501          * If we cloned to an opened newdev that already has called
502          * spec_assoc_vp_with_devi (SDIPSET set) then the association is
503          * already established.
504          */
505         if (!(newcsp->s_flag & SDIPSET)) {
506                 /*
507                  * Establish s_dip association for newdev.
508                  *
509                  * If we trusted the getinfo(9E) DDI_INFO_DEVT2INSTANCE
510                  * implementation of all cloning drivers  (SCLONE and SELFCLONE)
511                  * we would always use e_ddi_hold_devi_by_dev().  We know that
512                  * many drivers have had (still have?) problems with
513                  * DDI_INFO_DEVT2INSTANCE, so we try to minimize reliance by
514                  * detecting drivers that use QASSOCIATE (by looking down the
515                  * stream) and setting their s_dip association to NULL.
516                  */
517                 qassociate_used = 0;
518                 if (stp) {
519                         for (dq = stp->sd_wrq; dq; dq = dq->q_next) {
520                                 if (_RD(dq)->q_flag & _QASSOCIATED) {
521                                         qassociate_used = 1;
522                                         dip = NULL;
523                                         break;
```

```
524                                 }
525                         }
526                 }

528                 if (dip || qassociate_used) {
529                         spec_assoc_vp_with_devi(newvp, dip);
530                 } else {
531                         /* derive association from newdev */
532                         dip = e_ddi_hold_devi_by_dev(newdev, 0);
533                         spec_assoc_vp_with_devi(newvp, dip);
534                         if (dip)
535                                 ddi_release_devi(dip);
536                 }
537         }

539         SN_HOLD(newcsp);

541         /* deal with stream stuff */
542         if (stp != NULL) {
543                 LOCK_CSP(newcsp);       /* synchronize stream open/close */
544                 mutex_enter(&newcsp->s_lock);
545                 newcvp->v_stream = newvp->v_stream = stp;
546                 stp->sd_vnode = newcvp;
547                 stp->sd_strtab = STREAMSTAB(newmaj);
548                 mutex_exit(&newcsp->s_lock);
549                 UNLOCK_CSP(newcsp);
550         }

552         /* substitute the vnode */
553         SN_RELE(oldcsp);
554         VN_RELE(*vpp);
555         *vpp = newvp;

557         return (0);
558 }

560 static int
561 spec_open(struct vnode **vpp, int flag, struct cred *cr, caller_context_t *cc)
562 {
563         major_t maj;
564         dev_t dev, newdev;
565         struct vnode *vp, *cvp;
566         struct snode *sp, *csp;
567         struct stdata *stp;
568         dev_info_t *dip;
569         int error, type;
570         contract_t *ct = NULL;
571         int open_returns_eintr;
572         slock_ret_t spec_locksp_ret;


575         flag &= ~FCREAT;                /* paranoia */

577         vp = *vpp;
578         sp = VTOS(vp);
579         ASSERT((vp->v_type == VCHR) || (vp->v_type == VBLK));
580         if ((vp->v_type != VCHR) && (vp->v_type != VBLK))
581                 return (ENXIO);

583         /*
584          * If the VFS_NODEVICES bit was set for the mount,
585          * do not allow opens of special devices.
586          */
587         if (sp->s_realvp && (sp->s_realvp->v_vfsp->vfs_flag & VFS_NODEVICES))
588                 return (ENXIO);
```



```
590         newdev = dev = vp->v_rdev;

592         /*
593          * If we are opening a node that has not had spec_assoc_vp_with_devi
594          * called against it (mknod outside /devices or a non-dacf makespecvp
595          * node) then SDIPSET will not be set. In this case we call an
596          * interface which will reconstruct the path and lookup (drive attach)
597          * through devfs (e_ddi_hold_devi_by_dev -> e_ddi_hold_devi_by_path ->
598          * devfs_lookupname).  For support of broken drivers that don't call
599          * ddi_create_minor_node for all minor nodes in their instance space,
600          * we call interfaces that operates at the directory/devinfo
601          * (major/instance) level instead of to the leaf/minor node level.
602          * After finding and attaching the dip we associate it with the
603          * common specfs vnode (s_dip), which sets SDIPSET.  A DL_DETACH_REQ
604          * to style-2 stream driver may set s_dip to NULL with SDIPSET set.
605          *
606          * NOTE: Although e_ddi_hold_devi_by_dev takes a dev_t argument, its
607          * implementation operates at the major/instance level since it only
608          * need to return a dip.
609          */
610         cvp = sp->s_commonvp;
611         csp = VTOS(cvp);
612         if (!(csp->s_flag & SDIPSET)) {
613                 /* try to attach, return error if we fail */
614                 if ((dip = e_ddi_hold_devi_by_dev(dev, 0)) == NULL)
615                         return (ENXIO);

617                 /* associate dip with the common snode s_dip */
618                 spec_assoc_vp_with_devi(vp, dip);
619                 ddi_release_devi(dip);  /* from e_ddi_hold_devi_by_dev */
620         }

622         /* check if device fenced off */
623         if (S_ISFENCED(sp))
624                 return (ENXIO);

626 #ifdef  DEBUG
627         /* verify attach/open exclusion guarantee */
628         dip = csp->s_dip;
629         ASSERT((dip == NULL) || i_ddi_devi_attached(dip));
630 #endif  /* DEBUG */

632         if ((error = secpolicy_spec_open(cr, vp, flag)) != 0)
633                 return (error);

635         /* Verify existance of open(9E) implementation. */
636         maj = getmajor(dev);
637         if ((maj >= devcnt) ||
638             (devopsp[maj]->devo_cb_ops == NULL) ||
639             (devopsp[maj]->devo_cb_ops->cb_open == NULL))
640                 return (ENXIO);

642         /*
643          * split STREAMS vs. non-STREAMS
644          *
645          * If the device is a dual-personality device, then we might want
646          * to allow for a regular OTYP_BLK open.  If however it's strictly
647          * a pure STREAMS device, the cb_open entry point will be
648          * nodev() which returns ENXIO.  This does make this failure path
649          * somewhat longer, but such attempts to use OTYP_BLK with STREAMS
650          * devices should be exceedingly rare.  (Most of the time they will
651          * be due to programmer error.)
652          */
653         if ((vp->v_type == VCHR) && (STREAMSTAB(maj)))
654                 goto streams_open;
```

```
656 not_streams:
657         /*
658          * Wait for in progress last close to complete. This guarantees
659          * to the driver writer that we will never be in the drivers
660          * open and close on the same (dev_t, otype) at the same time.
661          * Open count already incremented (SN_HOLD) on non-zero return.
662          * The wait is interruptible by a signal if the driver sets the
663          * D_OPEN_RETURNS_EINTR cb_ops(9S) cb_flag or sets the
664          * ddi-open-returns-eintr(9P) property in its driver.conf.
665          */
666         if ((devopsp[maj]->devo_cb_ops->cb_flag & D_OPEN_RETURNS_EINTR) ||
667             (devnamesp[maj].dn_flags & DN_OPEN_RETURNS_EINTR))
668                 open_returns_eintr = 1;
669         else
670                 open_returns_eintr = 0;
671         while ((spec_locksp_ret = SYNCHOLD_CSP_SIG(csp, open_returns_eintr)) !=
672             SUCCESS) {
673                 if (spec_locksp_ret == INTR)
674                         return (EINTR);
675         }

677         /* non streams open */
678         type = (vp->v_type == VBLK ? OTYP_BLK : OTYP_CHR);
679         error = dev_open(&newdev, flag, type, cr);

681         /* deal with clone case */
682         if (error == 0 && dev != newdev) {
683                 error = spec_clone(vpp, newdev, vp->v_type, NULL);
684                 /*
685                  * bail on clone failure, further processing
686                  * results in undefined behaviors.
687                  */
688                 if (error != 0)
689                         return (error);
690                 sp = VTOS(*vpp);
691                 csp = VTOS(sp->s_commonvp);
692         }

694         /*
695          * create contracts only for userland opens
696          * Successful open and cloning is done at this point.
697          */
698         if (error == 0 && !(flag & FKLYR)) {
699                 int spec_type;
700                 spec_type = (STOV(csp)->v_type == VCHR) ? S_IFCHR : S_IFBLK;
701                 if (contract_device_open(newdev, spec_type, NULL) != 0) {
702                         error = EIO;
703                 }
704         }

706         if (error == 0) {
707                 sp->s_size = SPEC_SIZE(csp);

709                 if ((csp->s_flag & SNEEDCLOSE) == 0) {
710                         int nmaj = getmajor(newdev);
711                         mutex_enter(&csp->s_lock);
712                         /* successful open needs a close later */
713                         csp->s_flag |= SNEEDCLOSE;

715                         /*
716                          * Invalidate possible cached "unknown" size
717                          * established by a VOP_GETATTR while open was in
718                          * progress, and the driver might fail prop_op(9E).
719                          */
720                         if (((cvp->v_type == VCHR) && (csp->s_size == 0)) ||
721                             ((cvp->v_type == VBLK) &&
```



```
722                             (csp->s_size == UNKNOWN_SIZE)))
723                                 csp->s_flag &= ~SSIZEVALID;

725                         if (devopsp[nmaj]->devo_cb_ops->cb_flag & D_64BIT)
726                                 csp->s_flag |= SLOFFSET;
727                         if (devopsp[nmaj]->devo_cb_ops->cb_flag & D_U64BIT)
728                                 csp->s_flag |= SLOFFSET | SANYOFFSET;
729                         mutex_exit(&csp->s_lock);
730                 }
731                 return (0);
732         }

734         /*
735          * Open failed. If we missed a close operation because
736          * we were trying to get the device open and it is the
737          * last in progress open that is failing then call close.
738          *
739          * NOTE: Only non-streams open has this race condition.
740          */
741         mutex_enter(&csp->s_lock);
742         csp->s_count--;                 /* decrement open count : SN_RELE */
743         if ((csp->s_count == 0) &&      /* no outstanding open */
744             (csp->s_mapcnt == 0) &&     /* no mapping */
745             (csp->s_flag & SNEEDCLOSE)) { /* need a close */
746                 csp->s_flag &= ~(SNEEDCLOSE | SSIZEVALID);

748                 /* See comment in spec_close() */
749                 if (csp->s_flag & (SCLONE | SSELFCLONE))
750                         csp->s_flag &= ~SDIPSET;

752                 csp->s_flag |= SCLOSING;
753                 mutex_exit(&csp->s_lock);

755                 ASSERT(*vpp != NULL);
756                 (void) device_close(*vpp, flag, cr);

758                 mutex_enter(&csp->s_lock);
759                 csp->s_flag &= ~SCLOSING;
760                 mutex_exit(&csp->s_lock);
761         } else {
762                 mutex_exit(&csp->s_lock);
763         }
764         return (error);

766 streams_open:
767         /*
768          * Lock common snode to prevent any new clone opens on this
769          * stream while one is in progress. This is necessary since
770          * the stream currently associated with the clone device will
771          * not be part of it after the clone open completes. Unfortunately
772          * we don't know in advance if this is a clone
773          * device so we have to lock all opens.
774          *
775          * If we fail, it's because of an interrupt - EINTR return is an
776          * expected aspect of opening a stream so we don't need to check
777          * D_OPEN_RETURNS_EINTR. Open count already incremented (SN_HOLD)
778          * on non-zero return.
779          */
780         if (LOCKHOLD_CSP_SIG(csp) != SUCCESS)
781                 return (EINTR);

783         error = stropen(cvp, &newdev, flag, cr);
784         stp = cvp->v_stream;

786         /* deal with the clone case */
787         if ((error == 0) && (dev != newdev)) {
```

```
788                 vp->v_stream = cvp->v_stream = NULL;
789                 UNLOCK_CSP(csp);
790                 error = spec_clone(vpp, newdev, vp->v_type, stp);
791                 /*
792                  * bail on clone failure, further processing
793                  * results in undefined behaviors.
794                  */
795                 if (error != 0)
796                         return (error);
797                 sp = VTOS(*vpp);
798                 csp = VTOS(sp->s_commonvp);
799         } else if (error == 0) {
800                 vp->v_stream = stp;
801                 UNLOCK_CSP(csp);
802         }

804         /*
805          * create contracts only for userland opens
806          * Successful open and cloning is done at this point.
807          */
808         if (error == 0 && !(flag & FKLYR)) {
809                 /* STREAM is of type S_IFCHR */
810                 if (contract_device_open(newdev, S_IFCHR, &ct) != 0) {
811                         UNLOCK_CSP(csp);
812                         (void) spec_close(vp, flag, 1, 0, cr, cc);
813                         return (EIO);
814                 }
815         }

817         if (error == 0) {
818                 /* STREAMS devices don't have a size */
819                 sp->s_size = csp->s_size = 0;

821                 if (!(stp->sd_flag & STRISTTY) || (flag & FNOCTTY))
822                         return (0);

824                 /* try to allocate it as a controlling terminal */
825                 if (strctty(stp) != EINTR)
826                         return (0);

828                 /* strctty() was interrupted by a signal */
829                 if (ct) {
830                         /* we only create contracts for userland opens */
831                         ASSERT(ttoproc(curthread));
832                         (void) contract_abandon(ct, ttoproc(curthread), 0);
833                 }
834                 (void) spec_close(vp, flag, 1, 0, cr, cc);
835                 return (EINTR);
836         }

838         /*
839          * Deal with stropen failure.
840          *
841          * sd_flag in the stream head cannot change since the
842          * common snode is locked before the call to stropen().
843          */
844         if ((stp != NULL) && (stp->sd_flag & STREOPENFAIL)) {
845                 /*
846                  * Open failed part way through.
847                  */
848                 mutex_enter(&stp->sd_lock);
849                 stp->sd_flag &= ~STREOPENFAIL;
850                 mutex_exit(&stp->sd_lock);

852                 UNLOCK_CSP(csp);
853                 (void) spec_close(vp, flag, 1, 0, cr, cc);
```



```
854         } else {
855                 UNLOCK_CSP(csp);
856                 SN_RELE(csp);
857         }

859         /*
860          * Resolution for STREAMS vs. regular character device: If the
861          * STREAMS open(9e) returns ENOSTR, then try an ordinary device
862          * open instead.
863          */
864         if (error == ENOSTR) {
865                 goto not_streams;
866         }
867         return (error);
868 }

870 /*ARGSUSED2*/
871 static int
872 spec_close(
873         struct vnode    *vp,
874         int             flag,
875         int             count,
876         offset_t        offset,
877         struct cred     *cr,
878         caller_context_t *ct)
879 {
880         struct vnode *cvp;
881         struct snode *sp, *csp;
882         enum vtype type;
883         dev_t dev;
884         int error = 0;
885         int sysclone;

887         if (!(flag & FKLYR)) {
888                 /* this only applies to closes of devices from userland */
889                 cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
890                 cleanshares(vp, ttoproc(curthread)->p_pid);
891                 if (vp->v_stream)
892                         strclean(vp);
893         }
894         if (count > 1)
895                 return (0);

897         /* we allow close to succeed even if device is fenced off */
898         sp = VTOS(vp);
899         cvp = sp->s_commonvp;

901         dev = sp->s_dev;
902         type = vp->v_type;

904         ASSERT(type == VCHR || type == VBLK);

906         /*
907          * Prevent close/close and close/open races by serializing closes
908          * on this common snode. Clone opens are held up until after
909          * we have closed this device so the streams linkage is maintained
910          */
911         csp = VTOS(cvp);

913         LOCK_CSP(csp);
914         mutex_enter(&csp->s_lock);

916         csp->s_count--;                 /* one fewer open reference : SN_RELE */
917         sysclone = sp->s_flag & SCLONE;

919         /*
```

```
920          * Invalidate size on each close.
921          *
922          * XXX We do this on each close because we don't have interfaces that
923          * allow a driver to invalidate the size.  Since clearing this on each
924          * close this causes property overhead we skip /dev/null and
925          * /dev/zero to avoid degrading kenbus performance.
926          */
927         if (getmajor(dev) != mm_major)
928                 csp->s_flag &= ~SSIZEVALID;
929
930         /*
931          * Only call the close routine when the last open reference through
932          * any [s, v]node goes away.  This can be checked by looking at
933          * s_count on the common vnode.
934          */
935         if ((csp->s_count == 0) && (csp->s_mapcnt == 0)) {
936                 /* we don't need a close */
937                 csp->s_flag &= ~(SNEEDCLOSE | SSIZEVALID);
938
939                 /*
940                  * A cloning driver may open-clone to the same dev_t that we
941                  * are closing before spec_inactive destroys the common snode.
942                  * If this occurs the s_dip association needs to be reevaluated.
943                  * We clear SDIPSET to force reevaluation in this case.  When
944                  * reevaluation occurs (by spec_clone after open), if the
945                  * devinfo association has changed then the old association
946                  * will be released as the new association is established by
947                  * spec_assoc_vp_with_devi().
948                  */
949                 if (csp->s_flag & (SCLONE | SSELFCLONE))
950                         csp->s_flag &= ~SDIPSET;
951
952                 csp->s_flag |= SCLOSING;
953                 mutex_exit(&csp->s_lock);
954                 error = device_close(vp, flag, cr);
955
956                 /*
957                  * Decrement the devops held in clnopen()
958                  */
959                 if (sysclone) {
960                         ddi_rele_driver(getmajor(dev));
961                 }
962                 mutex_enter(&csp->s_lock);
963                 csp->s_flag &= ~SCLOSING;
964         }
965
966         UNLOCK_CSP_LOCK_HELD(csp);
967         mutex_exit(&csp->s_lock);
968
969         return (error);
970 }
971
972 /*ARGSUSED2*/
973 static int
974 spec_read(
975         struct vnode    *vp,
976         struct uio      *uiop,
977         int             ioflag,
978         struct cred     *cr,
979         caller_context_t *ct)
980 {
981         int error;
982         struct snode *sp = VTOS(vp);
983         dev_t dev = sp->s_dev;
984         size_t n;
985         ulong_t on;
```



```
986         u_offset_t bdevsize;
987         offset_t maxoff;
988         offset_t off;
989         struct vnode *blkvp;
990
991         ASSERT(vp->v_type == VCHR || vp->v_type == VBLK);
992
993         if (vp->v_stream) {
994                 ASSERT(vp->v_type == VCHR);
995                 smark(sp, SACC);
996                 return (strread(vp, uiop, cr));
997         }
998
999         if (uiop->uio_resid == 0)
1000                return (0);
1001
1002        /*
1003         * Plain old character devices that set D_U64BIT can have
1004         * unrestricted offsets.
1005         */
1006        maxoff = spec_maxoffset(vp);
1007        ASSERT(maxoff != -1 || vp->v_type == VCHR);
1008
1009        if (maxoff != -1 && (uiop->uio_loffset < 0 ||
1010            uiop->uio_loffset + uiop->uio_resid > maxoff))
1011                return (EINVAL);
1012
1013        if (vp->v_type == VCHR) {
1014                smark(sp, SACC);
1015                ASSERT(vp->v_stream == NULL);
1016                return (cdev_read(dev, uiop, cr));
1017        }
1018
1019        /*
1020         * Block device.
1021         */
1022        error = 0;
1023        blkvp = sp->s_commonvp;
1024        bdevsize = SPEC_SIZE(VTOS(blkvp));
1025
1026        do {
1027                caddr_t base;
1028                offset_t diff;
1029
1030                off = uiop->uio_loffset & (offset_t)MAXBMASK;
1031                on = (size_t)(uiop->uio_loffset & MAXBOFFSET);
1032                n = (size_t)MIN(MAXBSIZE - on, uiop->uio_resid);
1033                diff = bdevsize - uiop->uio_loffset;
1034
1035                if (diff <= 0)
1036                        break;
1037                if (diff < n)
1038                        n = (size_t)diff;
1039
1040                if (vpm_enable) {
1041                        error = vpm_data_copy(blkvp, (u_offset_t)(off + on),
1042                            n, uiop, 1, NULL, 0, S_READ);
1043                } else {
1044                        base = segmap_getmapflt(segkmap, blkvp,
1045                            (u_offset_t)(off + on), n, 1, S_READ);
1046
1047                        error = uiomove(base + on, n, UIO_READ, uiop);
1048                }
1049                if (!error) {
1050                        int flags = 0;
1051                        /*
```

```
1052                         * If we read a whole block, we won't need this
1053                         * buffer again soon.
1054                         */
1055                        if (n + on == MAXBSIZE)
1056                                flags = SM_DONTNEED | SM_FREE;
1057                        if (vpm_enable) {
1058                                error = vpm_sync_pages(blkvp, off, n, flags);
1059                        } else {
1060                                error = segmap_release(segkmap, base, flags);
1061                        }
1062                } else {
1063                        if (vpm_enable) {
1064                                (void) vpm_sync_pages(blkvp, off, n, 0);
1065                        } else {
1066                                (void) segmap_release(segkmap, base, 0);
1067                        }
1068                        if (bdevsize == UNKNOWN_SIZE) {
1069                                error = 0;
1070                                break;
1071                        }
1072                }
1073        } while (error == 0 && uiop->uio_resid > 0 && n != 0);

1075        return (error);
1076 }

1078 /*ARGSUSED*/
1079 static int
1080 spec_write(
1081        struct vnode *vp,
1082        struct uio *uiop,
1083        int ioflag,
1084        struct cred *cr,
1085        caller_context_t *ct)
1086 {
1087        int error;
1088        struct snode *sp = VTOS(vp);
1089        dev_t dev = sp->s_dev;
1090        size_t n;
1091        ulong_t on;
1092        u_offset_t bdevsize;
1093        offset_t maxoff;
1094        offset_t off;
1095        struct vnode *blkvp;

1097        ASSERT(vp->v_type == VCHR || vp->v_type == VBLK);

1099        if (vp->v_stream) {
1100                ASSERT(vp->v_type == VCHR);
1101                smark(sp, SUPD);
1102                return (strwrite(vp, uiop, cr));
1103        }

1105        /*
1106         * Plain old character devices that set D_U64BIT can have
1107         * unrestricted offsets.
1108         */
1109        maxoff = spec_maxoffset(vp);
1110        ASSERT(maxoff != -1 || vp->v_type == VCHR);

1112        if (maxoff != -1 && (uiop->uio_loffset < 0 ||
1113            uiop->uio_loffset + uiop->uio_resid > maxoff))
1114                return (EINVAL);

1116        if (vp->v_type == VCHR) {
1117                smark(sp, SUPD);
```



```
1118                ASSERT(vp->v_stream == NULL);
1119                return (cdev_write(dev, uiop, cr));
1120        }

1122        if (uiop->uio_resid == 0)
1123                return (0);

1125        error = 0;
1126        blkvp = sp->s_commonvp;
1127        bdevsize = SPEC_SIZE(VTOS(blkvp));

1129        do {
1130                int pagecreate;
1131                int newpage;
1132                caddr_t base;
1133                offset_t diff;

1135                off = uiop->uio_loffset & (offset_t)MAXBMASK;
1136                on = (ulong_t)(uiop->uio_loffset & MAXBOFFSET);
1137                n = (size_t)MIN(MAXBSIZE - on, uiop->uio_resid);
1138                pagecreate = 0;

1140                diff = bdevsize - uiop->uio_loffset;
1141                if (diff <= 0) {
1142                        error = ENXIO;
1143                        break;
1144                }
1145                if (diff < n)
1146                        n = (size_t)diff;

1148                /*
1149                 * Check to see if we can skip reading in the page
1150                 * and just allocate the memory.  We can do this
1151                 * if we are going to rewrite the entire mapping
1152                 * or if we are going to write to end of the device
1153                 * from the beginning of the mapping.
1154                 */
1155                if (n == MAXBSIZE || (on == 0 && (off + n) == bdevsize))
1156                        pagecreate = 1;

1158                newpage = 0;

1160                /*
1161                 * Touch the page and fault it in if it is not in core
1162                 * before segmap_getmapflt or vpm_data_copy can lock it.
1163                 * This is to avoid the deadlock if the buffer is mapped
1164                 * to the same file through mmap which we want to write.
1165                 */
1166                uio_prefaultpages((long)n, uiop);

1168                if (vpm_enable) {
1169                        error = vpm_data_copy(blkvp, (u_offset_t)(off + on),
1170                            n, uiop, !pagecreate, NULL, 0, S_WRITE);
1171                } else {
1172                        base = segmap_getmapflt(segkmap, blkvp,
1173                            (u_offset_t)(off + on), n, !pagecreate, S_WRITE);

1175                        /*
1176                         * segmap_pagecreate() returns 1 if it calls
1177                         * page_create_va() to allocate any pages.
1178                         */

1180                        if (pagecreate)
1181                                newpage = segmap_pagecreate(segkmap, base + on,
1182                                    n, 0);
```

```
1184                        error = uiomove(base + on, n, UIO_WRITE, uiop);
1185                }
1186
1187                if (!vpm_enable && pagecreate &&
1188                    uiop->uio_loffset <
1189                    P2ROUNDUP_TYPED(off + on + n, PAGESIZE, offset_t)) {
1190                        /*
1191                         * We created pages w/o initializing them completely,
1192                         * thus we need to zero the part that wasn't set up.
1193                         * This can happen if we write to the end of the device
1194                         * or if we had some sort of error during the uiomove.
1195                         */
1196                        long nzero;
1197                        offset_t nmoved;
1198
1199                        nmoved = (uiop->uio_loffset - (off + on));
1200                        if (nmoved < 0 || nmoved > n) {
1201                                panic("spec_write: nmoved bogus");
1202                                /*NOTREACHED*/
1203                        }
1204                        nzero = (long)P2ROUNDUP(on + n, PAGESIZE) -
1205                            (on + nmoved);
1206                        if (nzero < 0 || (on + nmoved + nzero > MAXBSIZE)) {
1207                                panic("spec_write: nzero bogus");
1208                                /*NOTREACHED*/
1209                        }
1210                        (void) kzero(base + on + nmoved, (size_t)nzero);
1211                }
1212
1213                /*
1214                 * Unlock the pages which have been allocated by
1215                 * page_create_va() in segmap_pagecreate().
1216                 */
1217                if (!vpm_enable && newpage)
1218                        segmap_pageunlock(segkmap, base + on,
1219                            (size_t)n, S_WRITE);
1220
1221                if (error == 0) {
1222                        int flags = 0;
1223
1224                        /*
1225                         * Force write back for synchronous write cases.
1226                         */
1227                        if (ioflag & (FSYNC|FDSYNC))
1228                                flags = SM_WRITE;
1229                        else if (n + on == MAXBSIZE || IS_SWAPVP(vp)) {
1230                                /*
1231                                 * Have written a whole block.
1232                                 * Start an asynchronous write and
1233                                 * mark the buffer to indicate that
1234                                 * it won't be needed again soon.
1235                                 * Push swap files here, since it
1236                                 * won't happen anywhere else.
1237                                 */
1238                                flags = SM_WRITE | SM_ASYNC | SM_DONTNEED;
1239                        }
1240                        smark(sp, SUPD|SCHG);
1241                        if (vpm_enable) {
1242                                error = vpm_sync_pages(blkvp, off, n, flags);
1243                        } else {
1244                                error = segmap_release(segkmap, base, flags);
1245                        }
1246                } else {
1247                        if (vpm_enable) {
1248                                (void) vpm_sync_pages(blkvp, off, n, SM_INVAL);
1249                        } else {
```



```
1250                                (void) segmap_release(segkmap, base, SM_INVAL);
1251                        }
1252                }
1253
1254        } while (error == 0 && uiop->uio_resid > 0 && n != 0);
1255
1256        return (error);
1257 }
1258
1259 /*ARGSUSED6*/
1260 static int
1261 spec_ioctl(struct vnode *vp, int cmd, intptr_t arg, int mode, struct cred *cr,
1262     int *rvalp, caller_context_t *ct)
1263 {
1264        struct snode *sp;
1265        dev_t dev;
1266        int error;
1267
1268        if (vp->v_type != VCHR)
1269                return (ENOTTY);
1270
1271        /*
1272         * allow ioctls() to go through even for fenced snodes, as they
1273         * may include unconfiguration operation - for example popping of
1274         * streams modules.
1275         */
1276
1277        sp = VTOS(vp);
1278        dev = sp->s_dev;
1279        if (vp->v_stream) {
1280                error = strioctl(vp, cmd, arg, mode, U_TO_K, cr, rvalp);
1281        } else {
1282                error = cdev_ioctl(dev, cmd, arg, mode, cr, rvalp);
1283        }
1284        return (error);
1285 }
1286
1287 static int
1288 spec_getattr(
1289        struct vnode *vp,
1290        struct vattr *vap,
1291        int flags,
1292        struct cred *cr,
1293        caller_context_t *ct)
1294 {
1295        int error;
1296        struct snode *sp;
1297        struct vnode *realvp;
1298
1299        /* With ATTR_COMM we will not get attributes from realvp */
1300        if (flags & ATTR_COMM) {
1301                sp = VTOS(vp);
1302                vp = sp->s_commonvp;
1303        }
1304        sp = VTOS(vp);
1305
1306        /* we want stat() to fail with ENXIO if the device is fenced off */
1307        if (S_ISFENCED(sp))
1308                return (ENXIO);
1309
1310        realvp = sp->s_realvp;
1311
1312        if (realvp == NULL) {
1313                static int snode_shift  = 0;
1314
1315                /*
```

```
1316                 * Calculate the amount of bitshift to a snode pointer which
1317                 * will still keep it unique.  See below.
1318                 */
1319                if (snode_shift == 0)
1320                        snode_shift = highbit(sizeof (struct snode));
1321                ASSERT(snode_shift > 0);

1323                /*
1324                 * No real vnode behind this one.  Fill in the fields
1325                 * from the snode.
1326                 *
1327                 * This code should be refined to return only the
1328                 * attributes asked for instead of all of them.
1329                 */
1330                vap->va_type = vp->v_type;
1331                vap->va_mode = 0;
1332                vap->va_uid = vap->va_gid = 0;
1333                vap->va_fsid = sp->s_fsid;

1335                /*
1336                 * If the va_nodeid is > MAX_USHORT, then i386 stats might
1337                 * fail. So we shift down the snode pointer to try and get
1338                 * the most uniqueness into 16-bits.
1339                 */
1340                vap->va_nodeid = ((ino64_t)(uintptr_t)sp >> snode_shift) &
1341                    0xFFFF;
1342                vap->va_nlink = 0;
1343                vap->va_rdev = sp->s_dev;

1345                /*
1346                 * va_nblocks is the number of 512 byte blocks used to store
1347                 * the mknod for the device, not the number of blocks on the
1348                 * device itself.  This is typically zero since the mknod is
1349                 * represented directly in the inode itself.
1350                 */
1351                vap->va_nblocks = 0;
1352        } else {
1353                error = VOP_GETATTR(realvp, vap, flags, cr, ct);
1354                if (error != 0)
1355                        return (error);
1356        }

1358        /* set the size from the snode */
1359        vap->va_size = SPEC_SIZE(VTOS(sp->s_commonvp));
1360        vap->va_blksize = MAXBSIZE;

1362        mutex_enter(&sp->s_lock);
1363        vap->va_atime.tv_sec = sp->s_atime;
1364        vap->va_mtime.tv_sec = sp->s_mtime;
1365        vap->va_ctime.tv_sec = sp->s_ctime;
1366        mutex_exit(&sp->s_lock);

1368        vap->va_atime.tv_nsec = 0;
1369        vap->va_mtime.tv_nsec = 0;
1370        vap->va_ctime.tv_nsec = 0;
1371        vap->va_seq = 0;

1373        return (0);
1374 }

1376 static int
1377 spec_setattr(
1378        struct vnode *vp,
1379        struct vattr *vap,
1380        int flags,
1381        struct cred *cr,
```



```
1382        caller_context_t *ct)
1383 {
1384        struct snode *sp = VTOS(vp);
1385        struct vnode *realvp;
1386        int error;

1388        /* fail with ENXIO if the device is fenced off */
1389        if (S_ISFENCED(sp))
1390                return (ENXIO);

1392        if (vp->v_type == VCHR && vp->v_stream && (vap->va_mask & AT_SIZE)) {
1393                /*
1394                 * 1135080:     O_TRUNC should have no effect on
1395                 *              named pipes and terminal devices.
1396                 */
1397                ASSERT(vap->va_mask == AT_SIZE);
1398                return (0);
1399        }

1401        if ((realvp = sp->s_realvp) == NULL)
1402                error = 0;      /* no real vnode to update */
1403        else
1404                error = VOP_SETATTR(realvp, vap, flags, cr, ct);
1405        if (error == 0) {
1406                /*
1407                 * If times were changed, update snode.
1408                 */
1409                mutex_enter(&sp->s_lock);
1410                if (vap->va_mask & AT_ATIME)
1411                        sp->s_atime = vap->va_atime.tv_sec;
1412                if (vap->va_mask & AT_MTIME) {
1413                        sp->s_mtime = vap->va_mtime.tv_sec;
1414                        sp->s_ctime = gethrestime_sec();
1415                }
1416                mutex_exit(&sp->s_lock);
1417        }
1418        return (error);
1419 }

1421 static int
1422 spec_access(
1423        struct vnode *vp,
1424        int mode,
1425        int flags,
1426        struct cred *cr,
1427        caller_context_t *ct)
1428 {
1429        struct vnode *realvp;
1430        struct snode *sp = VTOS(vp);

1432        /* fail with ENXIO if the device is fenced off */
1433        if (S_ISFENCED(sp))
1434                return (ENXIO);

1436        if ((realvp = sp->s_realvp) != NULL)
1437                return (VOP_ACCESS(realvp, mode, flags, cr, ct));
1438        else
1439                return (0);     /* Allow all access. */
1440 }

1442 /*
1443  * This can be called if creat or an open with O_CREAT is done on the root
1444  * of a lofs mount where the mounted entity is a special file.
1445  */
1446 /*ARGSUSED*/
1447 static int
```

```
1448 spec_create(
1449         struct vnode *dvp,
1450         char *name,
1451         vattr_t *vap,
1452         enum vcexcl excl,
1453         int mode,
1454         struct vnode **vpp,
1455         struct cred *cr,
1456         int flag,
1457         caller_context_t *ct,
1458         vsecattr_t *vsecp)
1459 {
1460         int error;
1461         struct snode *sp = VTOS(dvp);

1463         /* fail with ENXIO if the device is fenced off */
1464         if (S_ISFENCED(sp))
1465                 return (ENXIO);

1467         ASSERT(dvp && (dvp->v_flag & VROOT) && *name == '\0');
1468         if (excl == NONEXCL) {
1469                 if (mode && (error = spec_access(dvp, mode, 0, cr, ct)))
1470                         return (error);
1471                 VN_HOLD(dvp);
1472                 return (0);
1473         }
1474         return (EEXIST);
1475 }

1477 /*
1478  * In order to sync out the snode times without multi-client problems,
1479  * make sure the times written out are never earlier than the times
1480  * already set in the vnode.
1481  */
1482 static int
1483 spec_fsync(
1484         struct vnode *vp,
1485         int syncflag,
1486         struct cred *cr,
1487         caller_context_t *ct)
1488 {
1489         struct snode *sp = VTOS(vp);
1490         struct vnode *realvp;
1491         struct vnode *cvp;
1492         struct vattr va, vatmp;

1494         /* allow syncing even if device is fenced off */

1496         /* If times didn't change, don't flush anything. */
1497         mutex_enter(&sp->s_lock);
1498         if ((sp->s_flag & (SACC|SUPD|SCHG)) == 0 && vp->v_type != VBLK) {
1499                 mutex_exit(&sp->s_lock);
1500                 return (0);
1501         }
1502         sp->s_flag &= ~(SACC|SUPD|SCHG);
1503         mutex_exit(&sp->s_lock);
1504         cvp = sp->s_commonvp;
1505         realvp = sp->s_realvp;

1507         if (vp->v_type == VBLK && cvp != vp && vn_has_cached_data(cvp) &&
1508             (cvp->v_flag & VISSWAP) == 0)
1509                 (void) VOP_PUTPAGE(cvp, (offset_t)0, 0, 0, cr, ct);

1511         /*
1512          * For devices that support it, force write cache to stable storage.
1513          * We don't need the lock to check s_flags since we can treat
```



```
1514          * SNOFLUSH as a hint.
1515          */
1516         if ((vp->v_type == VBLK || vp->v_type == VCHR) &&
1517             !(sp->s_flag & SNOFLUSH)) {
1518                 int rval, rc;
1519                 struct dk_callback spec_callback;

1521                 spec_callback.dkc_flag = FLUSH_VOLATILE;
1522                 spec_callback.dkc_callback = NULL;

1524                 /* synchronous flush on volatile cache */
1525                 rc = cdev_ioctl(vp->v_rdev, DKIOCFLUSHWRITECACHE,
1526                     (intptr_t)&spec_callback, FNATIVE|FKIOCTL, cr, &rval);

1528                 if (rc == ENOTSUP || rc == ENOTTY) {
1529                         mutex_enter(&sp->s_lock);
1530                         sp->s_flag |= SNOFLUSH;
1531                         mutex_exit(&sp->s_lock);
1532                 }
1533         }

1535         /*
1536          * If no real vnode to update, don't flush anything.
1537          */
1538         if (realvp == NULL)
1539                 return (0);

1541         vatmp.va_mask = AT_ATIME|AT_MTIME;
1542         if (VOP_GETATTR(realvp, &vatmp, 0, cr, ct) == 0) {

1544                 mutex_enter(&sp->s_lock);
1545                 if (vatmp.va_atime.tv_sec > sp->s_atime)
1546                         va.va_atime = vatmp.va_atime;
1547                 else {
1548                         va.va_atime.tv_sec = sp->s_atime;
1549                         va.va_atime.tv_nsec = 0;
1550                 }
1551                 if (vatmp.va_mtime.tv_sec > sp->s_mtime)
1552                         va.va_mtime = vatmp.va_mtime;
1553                 else {
1554                         va.va_mtime.tv_sec = sp->s_mtime;
1555                         va.va_mtime.tv_nsec = 0;
1556                 }
1557                 mutex_exit(&sp->s_lock);

1559                 va.va_mask = AT_ATIME|AT_MTIME;
1560                 (void) VOP_SETATTR(realvp, &va, 0, cr, ct);
1561         }
1562         (void) VOP_FSYNC(realvp, syncflag, cr, ct);
1563         return (0);
1564 }

1566 /*ARGSUSED*/
1567 static void
1568 spec_inactive(struct vnode *vp, struct cred *cr, caller_context_t *ct)
1569 {
1570         struct snode *sp = VTOS(vp);
1571         struct vnode *cvp;
1572         struct vnode *rvp;

1574         /*
1575          * If no one has reclaimed the vnode, remove from the
1576          * cache now.
1577          */
1578         if (vp->v_count < 1) {
1579                 panic("spec_inactive: Bad v_count");
```

```
1580                /*NOTREACHED*/
1581        }
1582        mutex_enter(&stable_lock);

1584        mutex_enter(&vp->v_lock);
1585        /*
1586         * Drop the temporary hold by vn_rele now
1587         */
1588        if (--vp->v_count != 0) {
1589                mutex_exit(&vp->v_lock);
1590                mutex_exit(&stable_lock);
1591                return;
1592        }
1593        mutex_exit(&vp->v_lock);

1595        sdelete(sp);
1596        mutex_exit(&stable_lock);

1598        /* We are the sole owner of sp now */
1599        cvp = sp->s_commonvp;
1600        rvp = sp->s_realvp;

1602        if (rvp) {
1603                /*
1604                 * If the snode times changed, then update the times
1605                 * associated with the "realvp".
1606                 */
1607                if ((sp->s_flag & (SACC|SUPD|SCHG)) != 0) {

1609                        struct vattr va, vatmp;

1611                        mutex_enter(&sp->s_lock);
1612                        sp->s_flag &= ~(SACC|SUPD|SCHG);
1613                        mutex_exit(&sp->s_lock);
1614                        vatmp.va_mask = AT_ATIME|AT_MTIME;
1615                        /*
1616                         * The user may not own the device, but we
1617                         * want to update the attributes anyway.
1618                         */
1619                        if (VOP_GETATTR(rvp, &vatmp, 0, kcred, ct) == 0) {
1620                                if (vatmp.va_atime.tv_sec > sp->s_atime)
1621                                        va.va_atime = vatmp.va_atime;
1622                                else {
1623                                        va.va_atime.tv_sec = sp->s_atime;
1624                                        va.va_atime.tv_nsec = 0;
1625                                }
1626                                if (vatmp.va_mtime.tv_sec > sp->s_mtime)
1627                                        va.va_mtime = vatmp.va_mtime;
1628                                else {
1629                                        va.va_mtime.tv_sec = sp->s_mtime;
1630                                        va.va_mtime.tv_nsec = 0;
1631                                }

1633                                va.va_mask = AT_ATIME|AT_MTIME;
1634                                (void) VOP_SETATTR(rvp, &va, 0, kcred, ct);
1635                        }
1636                }
1637        }
1638        ASSERT(!vn_has_cached_data(vp));
1639        vn_invalid(vp);

1641        /* if we are sharing another file systems vfs, release it */
1642        if (vp->v_vfsp && (vp->v_vfsp != &spec_vfs))
1643                VFS_RELE(vp->v_vfsp);

1645        /* if we have a realvp, release the realvp */
```



```
1646        if (rvp)
1647                VN_RELE(rvp);

1649        /* if we have a common, release the common */
1650        if (cvp && (cvp != vp)) {
1651                VN_RELE(cvp);
1652 #ifdef DEBUG
1653        } else if (cvp) {
1654                /*
1655                 * if this is the last reference to a common vnode, any
1656                 * associated stream had better have been closed
1657                 */
1658                ASSERT(cvp == vp);
1659                ASSERT(cvp->v_stream == NULL);
1660 #endif /* DEBUG */
1661        }

1663        /*
1664         * if we have a hold on a devinfo node (established by
1665         * spec_assoc_vp_with_devi), release the hold
1666         */
1667        if (sp->s_dip)
1668                ddi_release_devi(sp->s_dip);

1670        /*
1671         * If we have an associated device policy, release it.
1672         */
1673        if (sp->s_plcy != NULL)
1674                dpfree(sp->s_plcy);

1676        /*
1677         * If all holds on the devinfo node are through specfs/devfs
1678         * and we just destroyed the last specfs node associated with the
1679         * device, then the devinfo node reference count should now be
1680         * zero.  We can't check this because there may be other holds
1681         * on the node from non file system sources: ddi_hold_devi_by_instance
1682         * for example.
1683         */
1684        kmem_cache_free(snode_cache, sp);
1685 }

1687 static int
1688 spec_fid(struct vnode *vp, struct fid *fidp, caller_context_t *ct)
1689 {
1690        struct vnode *realvp;
1691        struct snode *sp = VTOS(vp);

1693        if ((realvp = sp->s_realvp) != NULL)
1694                return (VOP_FID(realvp, fidp, ct));
1695        else
1696                return (EINVAL);
1697 }

1699 /*ARGSUSED1*/
1700 static int
1701 spec_seek(
1702        struct vnode *vp,
1703        offset_t ooff,
1704        offset_t *noffp,
1705        caller_context_t *ct)
1706 {
1707        offset_t maxoff = spec_maxoffset(vp);

1709        if (maxoff == -1 || *noffp <= maxoff)
1710                return (0);
1711        else
```

```
1712                return (EINVAL);
1713 }

1715 static int
1716 spec_frlock(
1717         struct vnode *vp,
1718         int             cmd,
1719         struct flock64  *bfp,
1720         int             flag,
1721         offset_t        offset,
1722         struct flk_callback *flk_cbp,
1723         struct cred     *cr,
1724         caller_context_t *ct)
1725 {
1726         struct snode *sp = VTOS(vp);
1727         struct snode *csp;

1729         csp = VTOS(sp->s_commonvp);
1730         /*
1731          * If file is being mapped, disallow frlock.
1732          */
1733         if (csp->s_mapcnt > 0)
1734                 return (EAGAIN);

1736         return (fs_frlock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
1737 }

1739 static int
1740 spec_realvp(struct vnode *vp, struct vnode **vpp, caller_context_t *ct)
1741 {
1742         struct vnode *rvp;

1744         if ((rvp = VTOS(vp)->s_realvp) != NULL) {
1745                 vp = rvp;
1746                 if (VOP_REALVP(vp, &rvp, ct) == 0)
1747                         vp = rvp;
1748         }

1750         *vpp = vp;
1751         return (0);
1752 }

1754 /*
1755  * Return all the pages from [off..off + len] in block
1756  * or character device.
1757  */
1758 /*ARGSUSED*/
1759 static int
1760 spec_getpage(
1761         struct vnode    *vp,
1762         offset_t        off,
1763         size_t          len,
1764         uint_t          *protp,
1765         page_t          *pl[],
1766         size_t          plsz,
1767         struct seg      *seg,
1768         caddr_t         addr,
1769         enum seg_rw     rw,
1770         struct cred     *cr,
1771         caller_context_t *ct)
1772 {
1773         struct snode *sp = VTOS(vp);
1774         int err;

1776         ASSERT(sp->s_commonvp == vp);
```



```
1778         /*
1779          * XXX  Given the above assertion, this might not do
1780          *      what is wanted here.
1781          */
1782         if (vp->v_flag & VNOMAP)
1783                 return (ENOSYS);
1784         TRACE_4(TR_FAC_SPECFS, TR_SPECFS_GETPAGE,
1785             "specfs getpage:vp %p off %llx len %ld snode %p",
1786             vp, off, len, sp);

1788         switch (vp->v_type) {
1789         case VBLK:
1790                 if (protp != NULL)
1791                         *protp = PROT_ALL;

1793                 if (((u_offset_t)off + len) > (SPEC_SIZE(sp) + PAGEOFFSET))
1794                         return (EFAULT);        /* beyond EOF */

1796                 err = pvn_getpages(spec_getpage, vp, (u_offset_t)off, len,
1797                     protp, pl, plsz, seg, addr, rw, cr);
  24                 if (len <= PAGESIZE)
  25                         err = spec_getapage(vp, (u_offset_t)off, len, protp, pl,
  26                             plsz, seg, addr, rw, cr);
  27                 else
  28                         err = pvn_getpages(spec_getapage, vp, (u_offset_t)off,
  29                             len, protp, pl, plsz, seg, addr, rw, cr);
1798                 break;

1800         case VCHR:
1801                 cmn_err(CE_NOTE, "spec_getpage called for character device. "
1802                     "Check any non-ON consolidation drivers");
1803                 err = 0;
1804                 pl[0] = (page_t *)0;
1805                 break;

1807         default:
1808                 panic("spec_getpage: bad v_type 0x%x", vp->v_type);
1809                 /*NOTREACHED*/
1810         }

1812         return (err);
1813 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   19018 Thu Jan  8 09:14:36 2015
new/usr/src/uts/common/fs/swapfs/swap_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 1991, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/buf.h>
#include <sys/cred.h>
#include <sys/errno.h>
#include <sys/vnode.h>
#include <sys/vfs_opreg.h>
#include <sys/cmn_err.h>
#include <sys/swap.h>
#include <sys/mman.h>
#include <sys/vmsystm.h>
#include <sys/vtrace.h>
#include <sys/debug.h>
#include <sys/sysmacros.h>
#include <sys/vm.h>

#include <sys/fs/swapnode.h>

#include <vm/seg.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <fs/fs_subr.h>

#include <vm/seg_kp.h>

/*
 * Define the routines within this file.
 */
static int      swap_getpage(struct vnode *vp, offset_t off, size_t len,
    uint_t *protp, struct page **plarr, size_t plsz, struct seg *seg,
    caddr_t addr, enum seg_rw rw, struct cred *cr, caller_context_t *ct);
static int      swap_putpage(struct vnode *vp, offset_t off, size_t len,
    int flags, struct cred *cr, caller_context_t *ct);
static void     swap_inactive(struct vnode *vp, struct cred *cr,
```



```
    caller_context_t *ct);
static void     swap_dispose(vnode_t *vp, page_t *pp, int fl, int dn,
    cred_t *cr, caller_context_t *ct);

static int      swap_getapage(struct vnode *vp, u_offset_t off, size_t len,
    uint_t *protp, page_t **plarr, size_t plsz,
    struct seg *seg, caddr_t addr, enum seg_rw rw, struct cred *cr);

int     swap_getconpage(struct vnode *vp, u_offset_t off, size_t len,
    uint_t *protp, page_t **plarr, size_t plsz, page_t *conpp,
    uint_t *pszc, spgcnt_t *nreloc, struct seg *seg, caddr_t addr,
    enum seg_rw rw, struct cred *cr);

static int      swap_putapage(struct vnode *vp, page_t *pp, u_offset_t *off,
    size_t *lenp, int flags, struct cred *cr);

const fs_operation_def_t swap_vnodeops_template[] = {
        VOPNAME_INACTIVE,       { .vop_inactive = swap_inactive },
        VOPNAME_GETPAGE,        { .vop_getpage = swap_getpage },
        VOPNAME_PUTPAGE,        { .vop_putpage = swap_putpage },
        VOPNAME_DISPOSE,        { .vop_dispose = swap_dispose },
        VOPNAME_SETFL,          { .error = fs_error },
        VOPNAME_POLL,           { .error = fs_error },
        VOPNAME_PATHCONF,       { .error = fs_error },
        VOPNAME_GETSECATTR,     { .error = fs_error },
        VOPNAME_SHRLOCK,        { .error = fs_error },
        NULL,                   NULL
};

vnodeops_t *swap_vnodeops;

/* ARGSUSED */
static void
swap_inactive(
        struct vnode *vp,
        struct cred *cr,
        caller_context_t *ct)
{
        SWAPFS_PRINT(SWAP_VOPS, "swap_inactive: vp %x\n", vp, 0, 0, 0, 0);
}

/*
 * Return all the pages from [off..off+len] in given file
 */
/*ARGSUSED*/
static int
swap_getpage(
        struct vnode *vp,
        offset_t off,
        size_t len,
        uint_t *protp,
        page_t *pl[],
        size_t plsz,
        struct seg *seg,
        caddr_t addr,
        enum seg_rw rw,
        struct cred *cr,
        caller_context_t *ct)
{
        int err;

        SWAPFS_PRINT(SWAP_VOPS, "swap_getpage: vp %p, off %llx, len %lx\n",
            (void *)vp, off, len, 0, 0);

        TRACE_3(TR_FAC_SWAPFS, TR_SWAPFS_GETPAGE,
            "swapfs getpage:vp %p off %llx len %ld",
```

```
126             (void *)vp, off, len);

128         return (pvn_getpages(swap_getapage, vp, (u_offset_t)off, len, protp,
129             pl, plsz, seg, addr, rw, cr));
 32         if (len <= PAGESIZE) {
 33                 err = swap_getapage(vp, (u_offset_t)off, len, protp, pl, plsz,
 34                     seg, addr, rw, cr);
 35         } else {
 36                 err = pvn_getpages(swap_getapage, vp, (u_offset_t)off, len,
 37                     protp, pl, plsz, seg, addr, rw, cr);
 38         }

 40         return (err);
130 }

132 /*
133  * Called from pvn_getpages to get a particular page.
 44  * Called from pvn_getpages or swap_getpage to get a particular page.
134  */
135 /*ARGSUSED*/
136 static int
137 swap_getapage(
138         struct vnode *vp,
139         u_offset_t off,
140         size_t len,
141         uint_t *protp,
142         page_t *pl[],
143         size_t plsz,
144         struct seg *seg,
145         caddr_t addr,
146         enum seg_rw rw,
147         struct cred *cr)
148 {
149         struct page *pp, *rpp;
150         int flags;
151         int err = 0;
152         struct vnode *pvp = NULL;
153         u_offset_t poff;
154         int flag_noreloc;
155         se_t lock;
156         extern int kcage_on;
157         int upgrade = 0;

159         SWAPFS_PRINT(SWAP_VOPS, "swap_getapage: vp %p, off %llx, len %lx\n",
160             vp, off, len, 0, 0);

162         /*
163          * Until there is a call-back mechanism to cause SEGKP
164          * pages to be unlocked, make them non-relocatable.
165          */
166         if (SEG_IS_SEGKP(seg))
167                 flag_noreloc = PG_NORELOC;
168         else
169                 flag_noreloc = 0;

171         if (protp != NULL)
172                 *protp = PROT_ALL;

174         lock = (rw == S_CREATE ? SE_EXCL : SE_SHARED);

176 again:
177         if (pp = page_lookup(vp, off, lock)) {
178                 /*
179                  * In very rare instances, a segkp page may have been
180                  * relocated outside of the kernel by the kernel cage
181                  * due to the window between page_unlock() and
```



```
182                  * VOP_PUTPAGE() in segkp_unlock().  Due to the
183                  * rareness of these occurances, the solution is to
184                  * relocate the page to a P_NORELOC page.
185                  */
186                 if (flag_noreloc != 0) {
187                         if (!PP_ISNORELOC(pp) && kcage_on) {
188                                 if (lock != SE_EXCL) {
189                                         upgrade = 1;
190                                         if (!page_tryupgrade(pp)) {
191                                                 page_unlock(pp);
192                                                 lock = SE_EXCL;
193                                                 goto again;
194                                         }
195                                 }

197                                 if (page_relocate_cage(&pp, &rpp) != 0)
198                                         panic("swap_getapage: "
199                                             "page_relocate_cage failed");

201                                 pp = rpp;
202                         }
203                 }

205                 if (pl) {
206                         if (upgrade)
207                                 page_downgrade(pp);

209                         pl[0] = pp;
210                         pl[1] = NULL;
211                 } else {
212                         page_unlock(pp);
213                 }
214         } else {
215                 pp = page_create_va(vp, off, PAGESIZE,
216                     PG_WAIT | PG_EXCL | flag_noreloc,
217                     seg, addr);
218                 /*
219                  * Someone raced in and created the page after we did the
220                  * lookup but before we did the create, so go back and
221                  * try to look it up again.
222                  */
223                 if (pp == NULL)
224                         goto again;
225                 if (rw != S_CREATE) {
226                         err = swap_getphysname(vp, off, &pvp, &poff);
227                         if (pvp) {
228                                 struct anon *ap;
229                                 kmutex_t *ahm;

231                                 flags = (pl == NULL ? B_ASYNC|B_READ : B_READ);
232                                 err = VOP_PAGEIO(pvp, pp, poff,
233                                     PAGESIZE, flags, cr, NULL);

235                                 if (!err) {
236                                         ahm = AH_MUTEX(vp, off);
237                                         mutex_enter(ahm);

239                                         ap = swap_anon(vp, off);
240                                         if (ap == NULL) {
241                                                 panic("swap_getapage:"
242                                                     " null anon");
243                                         }

245                                         if (ap->an_pvp == pvp &&
246                                             ap->an_poff == poff) {
247                                                 swap_phys_free(pvp, poff,
```

```
248                                             PAGESIZE);
249                                     ap->an_pvp = NULL;
250                                     ap->an_poff = NULL;
251                                     hat_setmod(pp);
252                                 }

254                                 mutex_exit(ahm);
255                             }
256                         } else {
257                             if (!err)
258                                 pagezero(pp, 0, PAGESIZE);

260                             /*
261                              * If it's a fault ahead, release page_io_lock
262                              * and SE_EXCL we grabbed in page_create_va
263                              *
264                              * If we are here, we haven't called VOP_PAGEIO
265                              * and thus calling pvn_read_done(pp, B_READ)
266                              * below may mislead that we tried i/o. Besides,
267                              * in case of async, pvn_read_done() should
268                              * not be called by *getpage()
269                              */
270                             if (pl == NULL) {
271                                 /*
272                                  * swap_getphysname can return error
273                                  * only when we are getting called from
274                                  * swapslot_free which passes non-NULL
275                                  * pl to VOP_GETPAGE.
276                                  */
277                                 ASSERT(err == 0);
278                                 page_io_unlock(pp);
279                                 page_unlock(pp);
280                             }
281                         }
282                 }

284                 ASSERT(pp != NULL);

286                 if (err && pl)
287                         pvn_read_done(pp, B_ERROR);

289                 if (!err && pl)
290                         pvn_plist_init(pp, pl, plsz, off, PAGESIZE, rw);
291         }
292         TRACE_3(TR_FAC_SWAPFS, TR_SWAPFS_GETAPAGE,
293             "swapfs getapage:pp %p vp %p off %llx", pp, vp, off);
294         return (err);
295 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   57268 Thu Jan  8 09:14:36 2015
new/usr/src/uts/common/fs/tmpfs/tmp_vnops.c
5382 pvn_getpages handles lengths <= PAGESIZE just fine
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc.  All rights reserved.
#endif /* ! codereview */
 */

#include <sys/types.h>
#include <sys/param.h>
#include <sys/t_lock.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/user.h>
#include <sys/time.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/fcntl.h>
#include <sys/flock.h>
#include <sys/kmem.h>
#include <sys/uio.h>
#include <sys/errno.h>
#include <sys/stat.h>
#include <sys/cred.h>
#include <sys/dirent.h>
#include <sys/pathname.h>
#include <sys/vmsystm.h>
#include <sys/fs/tmp.h>
#include <sys/fs/tmpnode.h>
#include <sys/mman.h>
#include <vm/hat.h>
#include <vm/seg_vn.h>
#include <vm/seg_map.h>
#include <vm/seg.h>
#include <vm/anon.h>
```



```
#include <vm/as.h>
#include <vm/page.h>
#include <vm/pvn.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/swap.h>
#include <sys/buf.h>
#include <sys/vm.h>
#include <sys/vtrace.h>
#include <sys/policy.h>
#include <fs/fs_subr.h>

static int      tmp_getapage(struct vnode *, u_offset_t, size_t, uint_t *,
        page_t **, size_t, struct seg *, caddr_t, enum seg_rw, struct cred *);
static int      tmp_putapage(struct vnode *, page_t *, u_offset_t *, size_t *,
        int, struct cred *);

/* ARGSUSED1 */
static int
tmp_open(struct vnode **vpp, int flag, struct cred *cred, caller_context_t *ct)
{
        /*
         * swapon to a tmpfs file is not supported so access
         * is denied on open if VISSWAP is set.
         */
        if ((*vpp)->v_flag & VISSWAP)
                return (EINVAL);
        return (0);
}

/* ARGSUSED1 */
static int
tmp_close(
        struct vnode *vp,
        int flag,
        int count,
        offset_t offset,
        struct cred *cred,
        caller_context_t *ct)
{
        cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
        cleanshares(vp, ttoproc(curthread)->p_pid);
        return (0);
}

/*
 * wrtmp does the real work of write requests for tmpfs.
 */
static int
wrtmp(
        struct tmount *tm,
        struct tmpnode *tp,
        struct uio *uio,
        struct cred *cr,
        struct caller_context *ct)
{
        pgcnt_t pageoffset;     /* offset in pages */
        ulong_t segmap_offset;  /* pagesize byte offset into segmap */
        caddr_t base;           /* base of segmap */
        ssize_t bytes;          /* bytes to uiomove */
        pfn_t pagenumber;       /* offset in pages into tmp file */
        struct vnode *vp;
        int error = 0;
        int     pagecreate;     /* == 1 if we allocated a page */
        int     newpage;
        rlim64_t limit = uio->uio_llimit;
```

```
128         long oresid = uio->uio_resid;
129         timestruc_t now;
130
131         long tn_size_changed = 0;
132         long old_tn_size;
133         long new_tn_size;
134
135         vp = TNTOV(tp);
136         ASSERT(vp->v_type == VREG);
137
138         TRACE_1(TR_FAC_TMPFS, TR_TMPFS_RWTMP_START,
139             "tmp_wrtmp_start:vp %p", vp);
140
141         ASSERT(RW_WRITE_HELD(&tp->tn_contents));
142         ASSERT(RW_WRITE_HELD(&tp->tn_rwlock));
143
144         if (MANDLOCK(vp, tp->tn_mode)) {
145                 rw_exit(&tp->tn_contents);
146                 /*
147                  * tmp_getattr ends up being called by chklock
148                  */
149                 error = chklock(vp, FWRITE, uio->uio_loffset, uio->uio_resid,
150                     uio->uio_fmode, ct);
151                 rw_enter(&tp->tn_contents, RW_WRITER);
152                 if (error != 0) {
153                         TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
154                             "tmp_wrtmp_end:vp %p error %d", vp, error);
155                         return (error);
156                 }
157         }
158
159         if (uio->uio_loffset < 0)
160                 return (EINVAL);
161
162         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
163                 limit = MAXOFFSET_T;
164
165         if (uio->uio_loffset >= limit) {
166                 proc_t *p = ttoproc(curthread);
167
168                 mutex_enter(&p->p_lock);
169                 (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE], p->p_rctls,
170                     p, RCA_UNSAFE_SIGINFO);
171                 mutex_exit(&p->p_lock);
172                 return (EFBIG);
173         }
174
175         if (uio->uio_loffset >= MAXOFF_T) {
176                 TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
177                     "tmp_wrtmp_end:vp %p error %d", vp, EINVAL);
178                 return (EFBIG);
179         }
180
181         if (uio->uio_resid == 0) {
182                 TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
183                     "tmp_wrtmp_end:vp %p error %d", vp, 0);
184                 return (0);
185         }
186
187         if (limit > MAXOFF_T)
188                 limit = MAXOFF_T;
189
190         do {
191                 long    offset;
192                 long    delta;
```



```
194                 offset = (long)uio->uio_offset;
195                 pageoffset = offset & PAGEOFFSET;
196                 /*
197                  * A maximum of PAGESIZE bytes of data is transferred
198                  * each pass through this loop
199                  */
200                 bytes = MIN(PAGESIZE - pageoffset, uio->uio_resid);
201
202                 if (offset + bytes >= limit) {
203                         if (offset >= limit) {
204                                 error = EFBIG;
205                                 goto out;
206                         }
207                         bytes = limit - offset;
208                 }
209                 pagenumber = btop(offset);
210
211                 /*
212                  * delta is the amount of anonymous memory
213                  * to reserve for the file.
214                  * We always reserve in pagesize increments so
215                  * unless we're extending the file into a new page,
216                  * we don't need to call tmp_resv.
217                  */
218                 delta = offset + bytes -
219                     P2ROUNDUP_TYPED(tp->tn_size, PAGESIZE, u_offset_t);
220                 if (delta > 0) {
221                         pagecreate = 1;
222                         if (tmp_resv(tm, tp, delta, pagecreate)) {
223                                 /*
224                                  * Log file system full in the zone that owns
225                                  * the tmpfs mount, as well as in the global
226                                  * zone if necessary.
227                                  */
228                                 zcmn_err(tm->tm_vfsp->vfs_zone->zone_id,
229                                     CE_WARN, "%s: File system full, "
230                                     "swap space limit exceeded",
231                                     tm->tm_mntpath);
232
233                                 if (tm->tm_vfsp->vfs_zone->zone_id !=
234                                     GLOBAL_ZONEID) {
235
236                                         vfs_t *vfs = tm->tm_vfsp;
237
238                                         zcmn_err(GLOBAL_ZONEID,
239                                             CE_WARN, "%s: File system full, "
240                                             "swap space limit exceeded",
241                                             vfs->vfs_vnodecovered->v_path);
242                                 }
243                                 error = ENOSPC;
244                                 break;
245                         }
246                         tmpnode_growmap(tp, (ulong_t)offset + bytes);
247                 }
248                 /* grow the file to the new length */
249                 if (offset + bytes > tp->tn_size) {
250                         tn_size_changed = 1;
251                         old_tn_size = tp->tn_size;
252                         /*
253                          * Postpone updating tp->tn_size until uiomove() is
254                          * done.
255                          */
256                         new_tn_size = offset + bytes;
257                 }
258                 if (bytes == PAGESIZE) {
259                         /*
```

```
260                          * Writing whole page so reading from disk
261                          * is a waste
262                          */
263                         pagecreate = 1;
264                 } else {
265                         pagecreate = 0;
266                 }
267                 /*
268                  * If writing past EOF or filling in a hole
269                  * we need to allocate an anon slot.
270                  */
271                 if (anon_get_ptr(tp->tn_anon, pagenumber) == NULL) {
272                         (void) anon_set_ptr(tp->tn_anon, pagenumber,
273                             anon_alloc(vp, ptob(pagenumber)), ANON_SLEEP);
274                         pagecreate = 1;
275                         tp->tn_nblocks++;
276                 }

278                 /*
279                  * We have to drop the contents lock to allow the VM
280                  * system to reacquire it in tmp_getpage()
281                  */
282                 rw_exit(&tp->tn_contents);

284                 /*
285                  * Touch the page and fault it in if it is not in core
286                  * before segmap_getmapflt or vpm_data_copy can lock it.
287                  * This is to avoid the deadlock if the buffer is mapped
288                  * to the same file through mmap which we want to write.
289                  */
290                 uio_prefaultpages((long)bytes, uio);

292                 newpage = 0;
293                 if (vpm_enable) {
294                         /*
295                          * Copy data. If new pages are created, part of
296                          * the page that is not written will be initizliazed
297                          * with zeros.
298                          */
299                         error = vpm_data_copy(vp, offset, bytes, uio,
300                             !pagecreate, &newpage, 1, S_WRITE);
301                 } else {
302                         /* Get offset within the segmap mapping */
303                         segmap_offset = (offset & PAGEMASK) & MAXBOFFSET;
304                         base = segmap_getmapflt(segkmap, vp,
305                             (offset &  MAXBMASK), PAGESIZE, !pagecreate,
306                             S_WRITE);
307                 }


310                 if (!vpm_enable && pagecreate) {
311                         /*
312                          * segmap_pagecreate() returns 1 if it calls
313                          * page_create_va() to allocate any pages.
314                          */
315                         newpage = segmap_pagecreate(segkmap,
316                             base + segmap_offset, (size_t)PAGESIZE, 0);
317                         /*
318                          * Clear from the beginning of the page to the starting
319                          * offset of the data.
320                          */
321                         if (pageoffset != 0)
322                                 (void) kzero(base + segmap_offset,
323                                     (size_t)pageoffset);
324                 }
```



```
326                 if (!vpm_enable) {
327                         error = uiomove(base + segmap_offset + pageoffset,
328                             (long)bytes, UIO_WRITE, uio);
329                 }

331                 if (!vpm_enable && pagecreate &&
332                     uio->uio_offset < P2ROUNDUP(offset + bytes, PAGESIZE)) {
333                         long    zoffset; /* zero from offset into page */
334                         /*
335                          * We created pages w/o initializing them completely,
336                          * thus we need to zero the part that wasn't set up.
337                          * This happens on most EOF write cases and if
338                          * we had some sort of error during the uiomove.
339                          */
340                         long nmoved;

342                         nmoved = uio->uio_offset - offset;
343                         ASSERT((nmoved + pageoffset) <= PAGESIZE);

345                         /*
346                          * Zero from the end of data in the page to the
347                          * end of the page.
348                          */
349                         if ((zoffset = pageoffset + nmoved) < PAGESIZE)
350                                 (void) kzero(base + segmap_offset + zoffset,
351                                     (size_t)PAGESIZE - zoffset);
352                 }

354                 /*
355                  * Unlock the pages which have been allocated by
356                  * page_create_va() in segmap_pagecreate()
357                  */
358                 if (!vpm_enable && newpage) {
359                         segmap_pageunlock(segkmap, base + segmap_offset,
360                             (size_t)PAGESIZE, S_WRITE);
361                 }

363                 if (error) {
364                         /*
365                          * If we failed on a write, we must
366                          * be sure to invalidate any pages that may have
367                          * been allocated.
368                          */
369                         if (vpm_enable) {
370                                 (void) vpm_sync_pages(vp, offset, PAGESIZE,
371                                     SM_INVAL);
372                         } else {
373                                 (void) segmap_release(segkmap, base, SM_INVAL);
374                         }
375                 } else {
376                         if (vpm_enable) {
377                                 error = vpm_sync_pages(vp, offset, PAGESIZE,
378                                     0);
379                         } else {
380                                 error = segmap_release(segkmap, base, 0);
381                         }
382                 }

384                 /*
385                  * Re-acquire contents lock.
386                  */
387                 rw_enter(&tp->tn_contents, RW_WRITER);

389                 /*
390                  * Update tn_size.
391                  */
```

```
392                 if (tn_size_changed)
393                         tp->tn_size = new_tn_size;

395                 /*
396                  * If the uiomove failed, fix up tn_size.
397                  */
398                 if (error) {
399                         if (tn_size_changed) {
400                                 /*
401                                  * The uiomove failed, and we
402                                  * allocated blocks,so get rid
403                                  * of them.
404                                  */
405                                 (void) tmpnode_trunc(tm, tp,
406                                     (ulong_t)old_tn_size);
407                         }
408                 } else {
409                         /*
410                          * XXX - Can this be out of the loop?
411                          */
412                         if ((tp->tn_mode & (S_IXUSR | S_IXGRP | S_IXOTH)) &&
413                             (tp->tn_mode & (S_ISUID | S_ISGID)) &&
414                             secpolicy_vnode_setid_retain(cr,
415                             (tp->tn_mode & S_ISUID) != 0 && tp->tn_uid == 0)) {
416                                 /*
417                                  * Clear Set-UID & Set-GID bits on
418                                  * successful write if not privileged
419                                  * and at least one of the execute bits
420                                  * is set.  If we always clear Set-GID,
421                                  * mandatory file and record locking is
422                                  * unuseable.
423                                  */
424                                 tp->tn_mode &= ~(S_ISUID | S_ISGID);
425                         }
426                         gethrestime(&now);
427                         tp->tn_mtime = now;
428                         tp->tn_ctime = now;
429                 }
430         } while (error == 0 && uio->uio_resid > 0 && bytes != 0);

432 out:
433         /*
434          * If we've already done a partial-write, terminate
435          * the write but return no error.
436          */
437         if (oresid != uio->uio_resid)
438                 error = 0;
439         TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
440             "tmp_wrtmp_end:vp %p error %d", vp, error);
441         return (error);
442 }

444 /*
445  * rdtmp does the real work of read requests for tmpfs.
446  */
447 static int
448 rdtmp(
449         struct tmount *tm,
450         struct tmpnode *tp,
451         struct uio *uio,
452         struct caller_context *ct)
453 {
454         ulong_t pageoffset;     /* offset in tmpfs file (uio_offset) */
455         ulong_t segmap_offset;  /* pagesize byte offset into segmap */
456         caddr_t base;           /* base of segmap */
457         ssize_t bytes;          /* bytes to uiomove */
```



```
458         struct vnode *vp;
459         int error;
460         long oresid = uio->uio_resid;

462 #if defined(lint)
463         tm = tm;
464 #endif
465         vp = TNTOV(tp);

467         TRACE_1(TR_FAC_TMPFS, TR_TMPFS_RWTMP_START, "tmp_rdtmp_start:vp %p",
468             vp);

470         ASSERT(RW_LOCK_HELD(&tp->tn_contents));

472         if (MANDLOCK(vp, tp->tn_mode)) {
473                 rw_exit(&tp->tn_contents);
474                 /*
475                  * tmp_getattr ends up being called by chklock
476                  */
477                 error = chklock(vp, FREAD, uio->uio_loffset, uio->uio_resid,
478                     uio->uio_fmode, ct);
479                 rw_enter(&tp->tn_contents, RW_READER);
480                 if (error != 0) {
481                         TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
482                             "tmp_rdtmp_end:vp %p error %d", vp, error);
483                         return (error);
484                 }
485         }
486         ASSERT(tp->tn_type == VREG);

488         if (uio->uio_loffset >= MAXOFF_T) {
489                 TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
490                     "tmp_rdtmp_end:vp %p error %d", vp, EINVAL);
491                 return (0);
492         }
493         if (uio->uio_loffset < 0)
494                 return (EINVAL);
495         if (uio->uio_resid == 0) {
496                 TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
497                     "tmp_rdtmp_end:vp %p error %d", vp, 0);
498                 return (0);
499         }

501         vp = TNTOV(tp);

503         do {
504                 long diff;
505                 long offset;

507                 offset = uio->uio_offset;
508                 pageoffset = offset & PAGEOFFSET;
509                 bytes = MIN(PAGESIZE - pageoffset, uio->uio_resid);

511                 diff = tp->tn_size - offset;

513                 if (diff <= 0) {
514                         error = 0;
515                         goto out;
516                 }
517                 if (diff < bytes)
518                         bytes = diff;

520                 /*
521                  * We have to drop the contents lock to allow the VM system
522                  * to reacquire it in tmp_getpage() should the uiomove cause a
523                  * pagefault.
```

```
524                  */
525                 rw_exit(&tp->tn_contents);
526
527                 if (vpm_enable) {
528                         /*
529                          * Copy data.
530                          */
531                         error = vpm_data_copy(vp, offset, bytes, uio, 1, NULL,
532                             0, S_READ);
533                 } else {
534                         segmap_offset = (offset & PAGEMASK) & MAXBOFFSET;
535                         base = segmap_getmapflt(segkmap, vp, offset & MAXBMASK,
536                             bytes, 1, S_READ);
537
538                         error = uiomove(base + segmap_offset + pageoffset,
539                             (long)bytes, UIO_READ, uio);
540                 }
541
542                 if (error) {
543                         if (vpm_enable) {
544                                 (void) vpm_sync_pages(vp, offset, PAGESIZE, 0);
545                         } else {
546                                 (void) segmap_release(segkmap, base, 0);
547                         }
548                 } else {
549                         if (vpm_enable) {
550                                 error = vpm_sync_pages(vp, offset, PAGESIZE,
551                                     0);
552                         } else {
553                                 error = segmap_release(segkmap, base, 0);
554                         }
555                 }
556
557                 /*
558                  * Re-acquire contents lock.
559                  */
560                 rw_enter(&tp->tn_contents, RW_READER);
561
562         } while (error == 0 && uio->uio_resid > 0);
563
564 out:
565         gethrestime(&tp->tn_atime);
566
567         /*
568          * If we've already done a partial read, terminate
569          * the read but return no error.
570          */
571         if (oresid != uio->uio_resid)
572                 error = 0;
573
574         TRACE_2(TR_FAC_TMPFS, TR_TMPFS_RWTMP_END,
575             "tmp_rdtmp_end:vp %x error %d", vp, error);
576         return (error);
577 }
578
579 /* ARGSUSED2 */
580 static int
581 tmp_read(struct vnode *vp, struct uio *uiop, int ioflag, cred_t *cred,
582     struct caller_context *ct)
583 {
584         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
585         struct tmount *tm = (struct tmount *)VTOTM(vp);
586         int error;
587
588         /*
589          * We don't currently support reading non-regular files
```



```
590          */
591         if (vp->v_type == VDIR)
592                 return (EISDIR);
593         if (vp->v_type != VREG)
594                 return (EINVAL);
595         /*
596          * tmp_rwlock should have already been called from layers above
597          */
598         ASSERT(RW_READ_HELD(&tp->tn_rwlock));
599
600         rw_enter(&tp->tn_contents, RW_READER);
601
602         error = rdtmp(tm, tp, uiop, ct);
603
604         rw_exit(&tp->tn_contents);
605
606         return (error);
607 }
608
609 static int
610 tmp_write(struct vnode *vp, struct uio *uiop, int ioflag, struct cred *cred,
611     struct caller_context *ct)
612 {
613         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
614         struct tmount *tm = (struct tmount *)VTOTM(vp);
615         int error;
616
617         /*
618          * We don't currently support writing to non-regular files
619          */
620         if (vp->v_type != VREG)
621                 return (EINVAL);        /* XXX EISDIR? */
622
623         /*
624          * tmp_rwlock should have already been called from layers above
625          */
626         ASSERT(RW_WRITE_HELD(&tp->tn_rwlock));
627
628         rw_enter(&tp->tn_contents, RW_WRITER);
629
630         if (ioflag & FAPPEND) {
631                 /*
632                  * In append mode start at end of file.
633                  */
634                 uiop->uio_loffset = tp->tn_size;
635         }
636
637         error = wrtmp(tm, tp, uiop, cred, ct);
638
639         rw_exit(&tp->tn_contents);
640
641         return (error);
642 }
643
644 /* ARGSUSED */
645 static int
646 tmp_ioctl(
647         struct vnode *vp,
648         int com,
649         intptr_t data,
650         int flag,
651         struct cred *cred,
652         int *rvalp,
653         caller_context_t *ct)
654 {
655         return (ENOTTY);
```

```
656 }

658 /* ARGSUSED2 */
659 static int
660 tmp_getattr(
661         struct vnode *vp,
662         struct vattr *vap,
663         int flags,
664         struct cred *cred,
665         caller_context_t *ct)
666 {
667         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
668         struct vnode *mvp;
669         struct vattr va;
670         int attrs = 1;

672         /*
673          * A special case to handle the root tnode on a diskless nfs
674          * client who may have had its uid and gid inherited
675          * from an nfs vnode with nobody ownership.  Likely the
676          * root filesystem. After nfs is fully functional the uid/gid
677          * may be mapable so ask again.
678          * vfsp can't get unmounted because we hold vp.
679          */
680         if (vp->v_flag & VROOT &&
681             (mvp = vp->v_vfsp->vfs_vnodecovered) != NULL) {
682                 mutex_enter(&tp->tn_tlock);
683                 if (tp->tn_uid == UID_NOBODY || tp->tn_gid == GID_NOBODY) {
684                         mutex_exit(&tp->tn_tlock);
685                         bzero(&va, sizeof (struct vattr));
686                         va.va_mask = AT_UID|AT_GID;
687                         attrs = VOP_GETATTR(mvp, &va, 0, cred, ct);
688                 } else {
689                         mutex_exit(&tp->tn_tlock);
690                 }
691         }
692         mutex_enter(&tp->tn_tlock);
693         if (attrs == 0) {
694                 tp->tn_uid = va.va_uid;
695                 tp->tn_gid = va.va_gid;
696         }
697         vap->va_type = vp->v_type;
698         vap->va_mode = tp->tn_mode & MODEMASK;
699         vap->va_uid = tp->tn_uid;
700         vap->va_gid = tp->tn_gid;
701         vap->va_fsid = tp->tn_fsid;
702         vap->va_nodeid = (ino64_t)tp->tn_nodeid;
703         vap->va_nlink = tp->tn_nlink;
704         vap->va_size = (u_offset_t)tp->tn_size;
705         vap->va_atime = tp->tn_atime;
706         vap->va_mtime = tp->tn_mtime;
707         vap->va_ctime = tp->tn_ctime;
708         vap->va_blksize = PAGESIZE;
709         vap->va_rdev = tp->tn_rdev;
710         vap->va_seq = tp->tn_seq;

712         /*
713          * XXX Holes are not taken into account.  We could take the time to
714          * run through the anon array looking for allocated slots...
715          */
716         vap->va_nblocks = (fsblkcnt64_t)btodb(ptob(btopr(vap->va_size)));
717         mutex_exit(&tp->tn_tlock);
718         return (0);
719 }

721 /*ARGSUSED4*/
```



```
722 static int
723 tmp_setattr(
724         struct vnode *vp,
725         struct vattr *vap,
726         int flags,
727         struct cred *cred,
728         caller_context_t *ct)
729 {
730         struct tmount *tm = (struct tmount *)VTOTM(vp);
731         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
732         int error = 0;
733         struct vattr *get;
734         long mask;

736         /*
737          * Cannot set these attributes
738          */
739         if ((vap->va_mask & AT_NOSET) || (vap->va_mask & AT_XVATTR))
740                 return (EINVAL);

742         mutex_enter(&tp->tn_tlock);

744         get = &tp->tn_attr;
745         /*
746          * Change file access modes. Must be owner or have sufficient
747          * privileges.
748          */
749         error = secpolicy_vnode_setattr(cred, vp, vap, get, flags, tmp_taccess,
750             tp);

752         if (error)
753                 goto out;

755         mask = vap->va_mask;

757         if (mask & AT_MODE) {
758                 get->va_mode &= S_IFMT;
759                 get->va_mode |= vap->va_mode & ~S_IFMT;
760         }

762         if (mask & AT_UID)
763                 get->va_uid = vap->va_uid;
764         if (mask & AT_GID)
765                 get->va_gid = vap->va_gid;
766         if (mask & AT_ATIME)
767                 get->va_atime = vap->va_atime;
768         if (mask & AT_MTIME)
769                 get->va_mtime = vap->va_mtime;

771         if (mask & (AT_UID | AT_GID | AT_MODE | AT_MTIME))
772                 gethrestime(&tp->tn_ctime);

774         if (mask & AT_SIZE) {
775                 ASSERT(vp->v_type != VDIR);

777                 /* Don't support large files. */
778                 if (vap->va_size > MAXOFF_T) {
779                         error = EFBIG;
780                         goto out;
781                 }
782                 mutex_exit(&tp->tn_tlock);

784                 rw_enter(&tp->tn_rwlock, RW_WRITER);
785                 rw_enter(&tp->tn_contents, RW_WRITER);
786                 error = tmpnode_trunc(tm, tp, (ulong_t)vap->va_size);
787                 rw_exit(&tp->tn_contents);
```

```
788                 rw_exit(&tp->tn_rwlock);

790                 if (error == 0 && vap->va_size == 0)
791                         vnevent_truncate(vp, ct);

793                 goto out1;
794         }
795 out:
796         mutex_exit(&tp->tn_tlock);
797 out1:
798         return (error);
799 }

801 /* ARGSUSED2 */
802 static int
803 tmp_access(
804         struct vnode *vp,
805         int mode,
806         int flags,
807         struct cred *cred,
808         caller_context_t *ct)
809 {
810         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
811         int error;

813         mutex_enter(&tp->tn_tlock);
814         error = tmp_taccess(tp, mode, cred);
815         mutex_exit(&tp->tn_tlock);
816         return (error);
817 }

819 /* ARGSUSED3 */
820 static int
821 tmp_lookup(
822         struct vnode *dvp,
823         char *nm,
824         struct vnode **vpp,
825         struct pathname *pnp,
826         int flags,
827         struct vnode *rdir,
828         struct cred *cred,
829         caller_context_t *ct,
830         int *direntflags,
831         pathname_t *realpnp)
832 {
833         struct tmpnode *tp = (struct tmpnode *)VTOTN(dvp);
834         struct tmpnode *ntp = NULL;
835         int error;


838         /* allow cd into @ dir */
839         if (flags & LOOKUP_XATTR) {
840                 struct tmpnode *xdp;
841                 struct tmount *tm;

843                 /*
844                  * don't allow attributes if not mounted XATTR support
845                  */
846                 if (!(dvp->v_vfsp->vfs_flag & VFS_XATTR))
847                         return (EINVAL);

849                 if (tp->tn_flags & ISXATTR)
850                         /* No attributes on attributes */
851                         return (EINVAL);

853                 rw_enter(&tp->tn_rwlock, RW_WRITER);
```



```
854                 if (tp->tn_xattrdp == NULL) {
855                         if (!(flags & CREATE_XATTR_DIR)) {
856                                 rw_exit(&tp->tn_rwlock);
857                                 return (ENOENT);
858                         }

860                         /*
861                          * No attribute directory exists for this
862                          * node - create the attr dir as a side effect
863                          * of this lookup.
864                          */

866                         /*
867                          * Make sure we have adequate permission...
868                          */

870                         if ((error = tmp_taccess(tp, VWRITE, cred)) != 0) {
871                                 rw_exit(&tp->tn_rwlock);
872                                 return (error);
873                         }

875                         xdp = tmp_memalloc(sizeof (struct tmpnode),
876                             TMP_MUSTHAVE);
877                         tm = VTOTM(dvp);
878                         tmpnode_init(tm, xdp, &tp->tn_attr, NULL);
879                         /*
880                          * Fix-up fields unique to attribute directories.
881                          */
882                         xdp->tn_flags = ISXATTR;
883                         xdp->tn_type = VDIR;
884                         if (tp->tn_type == VDIR) {
885                                 xdp->tn_mode = tp->tn_attr.va_mode;
886                         } else {
887                                 xdp->tn_mode = 0700;
888                                 if (tp->tn_attr.va_mode & 0040)
889                                         xdp->tn_mode |= 0750;
890                                 if (tp->tn_attr.va_mode & 0004)
891                                         xdp->tn_mode |= 0705;
892                         }
893                         xdp->tn_vnode->v_type = VDIR;
894                         xdp->tn_vnode->v_flag |= V_XATTRDIR;
895                         tdirinit(tp, xdp);
896                         tp->tn_xattrdp = xdp;
897                 } else {
898                         VN_HOLD(tp->tn_xattrdp->tn_vnode);
899                 }
900                 *vpp = TNTOV(tp->tn_xattrdp);
901                 rw_exit(&tp->tn_rwlock);
902                 return (0);
903         }

905         /*
906          * Null component name is a synonym for directory being searched.
907          */
908         if (*nm == '\0') {
909                 VN_HOLD(dvp);
910                 *vpp = dvp;
911                 return (0);
912         }
913         ASSERT(tp);

915         error = tdirlookup(tp, nm, &ntp, cred);

917         if (error == 0) {
918                 ASSERT(ntp);
919                 *vpp = TNTOV(ntp);
```

```
920                 /*
921                  * If vnode is a device return special vnode instead
922                  */
923                 if (IS_DEVVP(*vpp)) {
924                         struct vnode *newvp;
926                         newvp = specvp(*vpp, (*vpp)->v_rdev, (*vpp)->v_type,
927                             cred);
928                         VN_RELE(*vpp);
929                         *vpp = newvp;
930                 }
931         }
932         TRACE_4(TR_FAC_TMPFS, TR_TMPFS_LOOKUP,
933             "tmpfs lookup:vp %p name %s vpp %p error %d",
934             dvp, nm, vpp, error);
935         return (error);
936 }
938 /*ARGSUSED7*/
939 static int
940 tmp_create(
941         struct vnode *dvp,
942         char *nm,
943         struct vattr *vap,
944         enum vcexcl exclusive,
945         int mode,
946         struct vnode **vpp,
947         struct cred *cred,
948         int flag,
949         caller_context_t *ct,
950         vsecattr_t *vsecp)
951 {
952         struct tmpnode *parent;
953         struct tmount *tm;
954         struct tmpnode *self;
955         int error;
956         struct tmpnode *oldtp;
958 again:
959         parent = (struct tmpnode *)VTOTN(dvp);
960         tm = (struct tmount *)VTOTM(dvp);
961         self = NULL;
962         error = 0;
963         oldtp = NULL;
965         /* device files not allowed in ext. attr dirs */
966         if ((parent->tn_flags & ISXATTR) &&
967             (vap->va_type == VBLK || vap->va_type == VCHR ||
968             vap->va_type == VFIFO || vap->va_type == VDOOR ||
969             vap->va_type == VSOCK || vap->va_type == VPORT))
970                         return (EINVAL);
972         if (vap->va_type == VREG && (vap->va_mode & VSVTX)) {
973                 /* Must be privileged to set sticky bit */
974                 if (secpolicy_vnode_stky_modify(cred))
975                         vap->va_mode &= ~VSVTX;
976         } else if (vap->va_type == VNON) {
977                 return (EINVAL);
978         }
980         /*
981          * Null component name is a synonym for directory being searched.
982          */
983         if (*nm == '\0') {
984                 VN_HOLD(dvp);
985                 oldtp = parent;
```



```
986         } else {
987                 error = tdirlookup(parent, nm, &oldtp, cred);
988         }
990         if (error == 0) {       /* name found */
991                 boolean_t trunc = B_FALSE;
993                 ASSERT(oldtp);
995                 rw_enter(&oldtp->tn_rwlock, RW_WRITER);
997                 /*
998                  * if create/read-only an existing
999                  * directory, allow it
1000                 */
1001                if (exclusive == EXCL)
1002                        error = EEXIST;
1003                else if ((oldtp->tn_type == VDIR) && (mode & VWRITE))
1004                        error = EISDIR;
1005                else {
1006                        error = tmp_taccess(oldtp, mode, cred);
1007                }
1009                if (error) {
1010                        rw_exit(&oldtp->tn_rwlock);
1011                        tmpnode_rele(oldtp);
1012                        return (error);
1013                }
1014                *vpp = TNTOV(oldtp);
1015                if ((*vpp)->v_type == VREG && (vap->va_mask & AT_SIZE) &&
1016                    vap->va_size == 0) {
1017                        rw_enter(&oldtp->tn_contents, RW_WRITER);
1018                        (void) tmpnode_trunc(tm, oldtp, 0);
1019                        rw_exit(&oldtp->tn_contents);
1020                        trunc = B_TRUE;
1021                }
1022                rw_exit(&oldtp->tn_rwlock);
1023                if (IS_DEVVP(*vpp)) {
1024                        struct vnode *newvp;
1026                        newvp = specvp(*vpp, (*vpp)->v_rdev, (*vpp)->v_type,
1027                            cred);
1028                        VN_RELE(*vpp);
1029                        if (newvp == NULL) {
1030                                return (ENOSYS);
1031                        }
1032                        *vpp = newvp;
1033                }
1035                if (trunc)
1036                        vnevent_create(*vpp, ct);
1038                return (0);
1039        }
1041        if (error != ENOENT)
1042                return (error);
1044        rw_enter(&parent->tn_rwlock, RW_WRITER);
1045        error = tdirenter(tm, parent, nm, DE_CREATE,
1046            (struct tmpnode *)NULL, (struct tmpnode *)NULL,
1047            vap, &self, cred, ct);
1048        rw_exit(&parent->tn_rwlock);
1050        if (error) {
1051                if (self)
```

```
1052                        tmpnode_rele(self);

1054                if (error == EEXIST) {
1055                        /*
1056                         * This means that the file was created sometime
1057                         * after we checked and did not find it and when
1058                         * we went to create it.
1059                         * Since creat() is supposed to truncate a file
1060                         * that already exits go back to the begining
1061                         * of the function. This time we will find it
1062                         * and go down the tmp_trunc() path
1063                         */
1064                        goto again;
1065                }
1066                return (error);
1067        }

1069        *vpp = TNTOV(self);

1071        if (!error && IS_DEVVP(*vpp)) {
1072                struct vnode *newvp;

1074                newvp = specvp(*vpp, (*vpp)->v_rdev, (*vpp)->v_type, cred);
1075                VN_RELE(*vpp);
1076                if (newvp == NULL)
1077                        return (ENOSYS);
1078                *vpp = newvp;
1079        }
1080        TRACE_3(TR_FAC_TMPFS, TR_TMPFS_CREATE,
1081            "tmpfs create:dvp %p nm %s vpp %p", dvp, nm, vpp);
1082        return (0);
1083 }

1085 /* ARGSUSED3 */
1086 static int
1087 tmp_remove(
1088        struct vnode *dvp,
1089        char *nm,
1090        struct cred *cred,
1091        caller_context_t *ct,
1092        int flags)
1093 {
1094        struct tmpnode *parent = (struct tmpnode *)VTOTN(dvp);
1095        int error;
1096        struct tmpnode *tp = NULL;

1098        error = tdirlookup(parent, nm, &tp, cred);
1099        if (error)
1100                return (error);

1102        ASSERT(tp);
1103        rw_enter(&parent->tn_rwlock, RW_WRITER);
1104        rw_enter(&tp->tn_rwlock, RW_WRITER);

1106        if (tp->tn_type != VDIR ||
1107            (error = secpolicy_fs_linkdir(cred, dvp->v_vfsp)) == 0)
1108                error = tdirdelete(parent, tp, nm, DR_REMOVE, cred);

1110        rw_exit(&tp->tn_rwlock);
1111        rw_exit(&parent->tn_rwlock);
1112        vnevent_remove(TNTOV(tp), dvp, nm, ct);
1113        tmpnode_rele(tp);

1115        TRACE_3(TR_FAC_TMPFS, TR_TMPFS_REMOVE,
1116            "tmpfs remove:dvp %p nm %s error %d", dvp, nm, error);
1117        return (error);
```



```
1118 }

1120 /* ARGSUSED4 */
1121 static int
1122 tmp_link(
1123        struct vnode *dvp,
1124        struct vnode *srcvp,
1125        char *tnm,
1126        struct cred *cred,
1127        caller_context_t *ct,
1128        int flags)
1129 {
1130        struct tmpnode *parent;
1131        struct tmpnode *from;
1132        struct tmount *tm = (struct tmount *)VTOTM(dvp);
1133        int error;
1134        struct tmpnode *found = NULL;
1135        struct vnode *realvp;

1137        if (VOP_REALVP(srcvp, &realvp, ct) == 0)
1138                srcvp = realvp;

1140        parent = (struct tmpnode *)VTOTN(dvp);
1141        from = (struct tmpnode *)VTOTN(srcvp);

1143        if ((srcvp->v_type == VDIR &&
1144            secpolicy_fs_linkdir(cred, dvp->v_vfsp)) ||
1145            (from->tn_uid != crgetuid(cred) && secpolicy_basic_link(cred)))
1146                return (EPERM);

1148        /*
1149         * Make sure link for extended attributes is valid
1150         * We only support hard linking of xattr's in xattrdir to an xattrdir
1151         */
1152        if ((from->tn_flags & ISXATTR) != (parent->tn_flags & ISXATTR))
1153                return (EINVAL);

1155        error = tdirlookup(parent, tnm, &found, cred);
1156        if (error == 0) {
1157                ASSERT(found);
1158                tmpnode_rele(found);
1159                return (EEXIST);
1160        }

1162        if (error != ENOENT)
1163                return (error);

1165        rw_enter(&parent->tn_rwlock, RW_WRITER);
1166        error = tdirenter(tm, parent, tnm, DE_LINK, (struct tmpnode *)NULL,
1167            from, NULL, (struct tmpnode **)NULL, cred, ct);
1168        rw_exit(&parent->tn_rwlock);
1169        if (error == 0) {
1170                vnevent_link(srcvp, ct);
1171        }
1172        return (error);
1173 }

1175 /* ARGSUSED5 */
1176 static int
1177 tmp_rename(
1178        struct vnode *odvp,     /* source parent vnode */
1179        char *onm,              /* source name */
1180        struct vnode *ndvp,     /* destination parent vnode */
1181        char *nnm,              /* destination name */
1182        struct cred *cred,
1183        caller_context_t *ct,
```

```
1184         int flags)
1185 {
1186         struct tmpnode *fromparent;
1187         struct tmpnode *toparent;
1188         struct tmpnode *fromtp = NULL;  /* source tmpnode */
1189         struct tmount *tm = (struct tmount *)VTOTM(odvp);
1190         int error;
1191         int samedir = 0;        /* set if odvp == ndvp */
1192         struct vnode *realvp;

1194         if (VOP_REALVP(ndvp, &realvp, ct) == 0)
1195                 ndvp = realvp;

1197         fromparent = (struct tmpnode *)VTOTN(odvp);
1198         toparent = (struct tmpnode *)VTOTN(ndvp);

1200         if ((fromparent->tn_flags & ISXATTR) != (toparent->tn_flags & ISXATTR))
1201                 return (EINVAL);

1203         mutex_enter(&tm->tm_renamelck);

1205         /*
1206          * Look up tmpnode of file we're supposed to rename.
1207          */
1208         error = tdirlookup(fromparent, onm, &fromtp, cred);
1209         if (error) {
1210                 mutex_exit(&tm->tm_renamelck);
1211                 return (error);
1212         }

1214         /*
1215          * Make sure we can delete the old (source) entry.  This
1216          * requires write permission on the containing directory.  If
1217          * that directory is "sticky" it requires further checks.
1218          */
1219         if (((error = tmp_taccess(fromparent, VWRITE, cred)) != 0) ||
1220             (error = tmp_sticky_remove_access(fromparent, fromtp, cred)) != 0)
1221                 goto done;

1223         /*
1224          * Check for renaming to or from '.' or '..' or that
1225          * fromtp == fromparent
1226          */
1227         if ((onm[0] == '.' &&
1228             (onm[1] == '\0' || (onm[1] == '.' && onm[2] == '\0'))) ||
1229             (nnm[0] == '.' &&
1230             (nnm[1] == '\0' || (nnm[1] == '.' && nnm[2] == '\0'))) ||
1231             (fromparent == fromtp)) {
1232                 error = EINVAL;
1233                 goto done;
1234         }

1236         samedir = (fromparent == toparent);
1237         /*
1238          * Make sure we can search and rename into the new
1239          * (destination) directory.
1240          */
1241         if (!samedir) {
1242                 error = tmp_taccess(toparent, VEXEC|VWRITE, cred);
1243                 if (error)
1244                         goto done;
1245         }

1247         /*
1248          * Link source to new target
1249          */
```



```
1250         rw_enter(&toparent->tn_rwlock, RW_WRITER);
1251         error = tdirenter(tm, toparent, nnm, DE_RENAME,
1252             fromparent, fromtp, (struct vattr *)NULL,
1253             (struct tmpnode **)NULL, cred, ct);
1254         rw_exit(&toparent->tn_rwlock);

1256         if (error) {
1257                 /*
1258                  * ESAME isn't really an error; it indicates that the
1259                  * operation should not be done because the source and target
1260                  * are the same file, but that no error should be reported.
1261                  */
1262                 if (error == ESAME)
1263                         error = 0;
1264                 goto done;
1265         }
1266         vnevent_rename_src(TNTOV(fromtp), odvp, onm, ct);

1268         /*
1269          * Notify the target directory if not same as
1270          * source directory.
1271          */
1272         if (ndvp != odvp) {
1273                 vnevent_rename_dest_dir(ndvp, ct);
1274         }

1276         /*
1277          * Unlink from source.
1278          */
1279         rw_enter(&fromparent->tn_rwlock, RW_WRITER);
1280         rw_enter(&fromtp->tn_rwlock, RW_WRITER);

1282         error = tdirdelete(fromparent, fromtp, onm, DR_RENAME, cred);

1284         /*
1285          * The following handles the case where our source tmpnode was
1286          * removed before we got to it.
1287          *
1288          * XXX We should also cleanup properly in the case where tdirdelete
1289          * fails for some other reason.  Currently this case shouldn't happen.
1290          * (see 1184991).
1291          */
1292         if (error == ENOENT)
1293                 error = 0;

1295         rw_exit(&fromtp->tn_rwlock);
1296         rw_exit(&fromparent->tn_rwlock);
1297 done:
1298         tmpnode_rele(fromtp);
1299         mutex_exit(&tm->tm_renamelck);

1301         TRACE_5(TR_FAC_TMPFS, TR_TMPFS_RENAME,
1302             "tmpfs rename:ovp %p onm %s nvp %p nnm %s error %d", odvp, onm,
1303             ndvp, nnm, error);
1304         return (error);
1305 }

1307 /* ARGSUSED5 */
1308 static int
1309 tmp_mkdir(
1310         struct vnode *dvp,
1311         char *nm,
1312         struct vattr *va,
1313         struct vnode **vpp,
1314         struct cred *cred,
1315         caller_context_t *ct,
```

```
1316         int flags,
1317         vsecattr_t *vsecp)
1318 {
1319         struct tmpnode *parent = (struct tmpnode *)VTOTN(dvp);
1320         struct tmpnode *self = NULL;
1321         struct tmount *tm = (struct tmount *)VTOTM(dvp);
1322         int error;

1324         /* no new dirs allowed in xattr dirs */
1325         if (parent->tn_flags & ISXATTR)
1326                 return (EINVAL);

1328         /*
1329          * Might be dangling directory.  Catch it here,
1330          * because a ENOENT return from tdirlookup() is
1331          * an "o.k. return".
1332          */
1333         if (parent->tn_nlink == 0)
1334                 return (ENOENT);

1336         error = tdirlookup(parent, nm, &self, cred);
1337         if (error == 0) {
1338                 ASSERT(self);
1339                 tmpnode_rele(self);
1340                 return (EEXIST);
1341         }
1342         if (error != ENOENT)
1343                 return (error);

1345         rw_enter(&parent->tn_rwlock, RW_WRITER);
1346         error = tdirenter(tm, parent, nm, DE_MKDIR, (struct tmpnode *)NULL,
1347             (struct tmpnode *)NULL, va, &self, cred, ct);
1348         if (error) {
1349                 rw_exit(&parent->tn_rwlock);
1350                 if (self)
1351                         tmpnode_rele(self);
1352                 return (error);
1353         }
1354         rw_exit(&parent->tn_rwlock);
1355         *vpp = TNTOV(self);
1356         return (0);
1357 }

1359 /* ARGSUSED4 */
1360 static int
1361 tmp_rmdir(
1362         struct vnode *dvp,
1363         char *nm,
1364         struct vnode *cdir,
1365         struct cred *cred,
1366         caller_context_t *ct,
1367         int flags)
1368 {
1369         struct tmpnode *parent = (struct tmpnode *)VTOTN(dvp);
1370         struct tmpnode *self = NULL;
1371         struct vnode *vp;
1372         int error = 0;

1374         /*
1375          * Return error when removing . and ..
1376          */
1377         if (strcmp(nm, ".") == 0)
1378                 return (EINVAL);
1379         if (strcmp(nm, "..") == 0)
1380                 return (EEXIST); /* Should be ENOTEMPTY */
1381         error = tdirlookup(parent, nm, &self, cred);
```



```
1382         if (error)
1383                 return (error);

1385         rw_enter(&parent->tn_rwlock, RW_WRITER);
1386         rw_enter(&self->tn_rwlock, RW_WRITER);

1388         vp = TNTOV(self);
1389         if (vp == dvp || vp == cdir) {
1390                 error = EINVAL;
1391                 goto done1;
1392         }
1393         if (self->tn_type != VDIR) {
1394                 error = ENOTDIR;
1395                 goto done1;
1396         }

1398         mutex_enter(&self->tn_tlock);
1399         if (self->tn_nlink > 2) {
1400                 mutex_exit(&self->tn_tlock);
1401                 error = EEXIST;
1402                 goto done1;
1403         }
1404         mutex_exit(&self->tn_tlock);

1406         if (vn_vfswlock(vp)) {
1407                 error = EBUSY;
1408                 goto done1;
1409         }
1410         if (vn_mountedvfs(vp) != NULL) {
1411                 error = EBUSY;
1412                 goto done;
1413         }

1415         /*
1416          * Check for an empty directory
1417          * i.e. only includes entries for "." and ".."
1418          */
1419         if (self->tn_dirents > 2) {
1420                 error = EEXIST;         /* SIGH should be ENOTEMPTY */
1421                 /*
1422                  * Update atime because checking tn_dirents is logically
1423                  * equivalent to reading the directory
1424                  */
1425                 gethrestime(&self->tn_atime);
1426                 goto done;
1427         }

1429         error = tdirdelete(parent, self, nm, DR_RMDIR, cred);
1430 done:
1431         vn_vfsunlock(vp);
1432 done1:
1433         rw_exit(&self->tn_rwlock);
1434         rw_exit(&parent->tn_rwlock);
1435         vnevent_rmdir(TNTOV(self), dvp, nm, ct);
1436         tmpnode_rele(self);

1438         return (error);
1439 }

1441 /* ARGSUSED2 */
1442 static int
1443 tmp_readdir(
1444         struct vnode *vp,
1445         struct uio *uiop,
1446         struct cred *cred,
1447         int *eofp,
```

```
1448         caller_context_t *ct,
1449         int flags)
1450 {
1451         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
1452         struct tdirent *tdp;
1453         int error = 0;
1454         size_t namelen;
1455         struct dirent64 *dp;
1456         ulong_t offset;
1457         ulong_t total_bytes_wanted;
1458         long outcount = 0;
1459         long bufsize;
1460         int reclen;
1461         caddr_t outbuf;

1463         if (uiop->uio_loffset >= MAXOFF_T) {
1464                 if (eofp)
1465                         *eofp = 1;
1466                 return (0);
1467         }
1468         /*
1469          * assuming system call has already called tmp_rwlock
1470          */
1471         ASSERT(RW_READ_HELD(&tp->tn_rwlock));

1473         if (uiop->uio_iovcnt != 1)
1474                 return (EINVAL);

1476         if (vp->v_type != VDIR)
1477                 return (ENOTDIR);

1479         /*
1480          * There's a window here where someone could have removed
1481          * all the entries in the directory after we put a hold on the
1482          * vnode but before we grabbed the rwlock.  Just return.
1483          */
1484         if (tp->tn_dir == NULL) {
1485                 if (tp->tn_nlink) {
1486                         panic("empty directory 0x%p", (void *)tp);
1487                         /*NOTREACHED*/
1488                 }
1489                 return (0);
1490         }

1492         /*
1493          * Get space for multiple directory entries
1494          */
1495         total_bytes_wanted = uiop->uio_iov->iov_len;
1496         bufsize = total_bytes_wanted + sizeof (struct dirent64);
1497         outbuf = kmem_alloc(bufsize, KM_SLEEP);

1499         dp = (struct dirent64 *)outbuf;


1502         offset = 0;
1503         tdp = tp->tn_dir;
1504         while (tdp) {
1505                 namelen = strlen(tdp->td_name); /* no +1 needed */
1506                 offset = tdp->td_offset;
1507                 if (offset >= uiop->uio_offset) {
1508                         reclen = (int)DIRENT64_RECLEN(namelen);
1509                         if (outcount + reclen > total_bytes_wanted) {
1510                                 if (!outcount)
1511                                         /*
1512                                          * Buffer too small for any entries.
1513                                          */
```



```
1514                                         error = EINVAL;
1515                                 break;
1516                         }
1517                         ASSERT(tdp->td_tmpnode != NULL);

1519                         /* use strncpy(9f) to zero out uninitialized bytes */

1521                         (void) strncpy(dp->d_name, tdp->td_name,
1522                             DIRENT64_NAMELEN(reclen));
1523                         dp->d_reclen = (ushort_t)reclen;
1524                         dp->d_ino = (ino64_t)tdp->td_tmpnode->tn_nodeid;
1525                         dp->d_off = (offset_t)tdp->td_offset + 1;
1526                         dp = (struct dirent64 *)
1527                             ((uintptr_t)dp + dp->d_reclen);
1528                         outcount += reclen;
1529                         ASSERT(outcount <= bufsize);
1530                 }
1531                 tdp = tdp->td_next;
1532         }

1534         if (!error)
1535                 error = uiomove(outbuf, outcount, UIO_READ, uiop);

1537         if (!error) {
1538                 /* If we reached the end of the list our offset */
1539                 /* should now be just past the end. */
1540                 if (!tdp) {
1541                         offset += 1;
1542                         if (eofp)
1543                                 *eofp = 1;
1544                 } else if (eofp)
1545                         *eofp = 0;
1546                 uiop->uio_offset = offset;
1547         }
1548         gethrestime(&tp->tn_atime);
1549         kmem_free(outbuf, bufsize);
1550         return (error);
1551 }

1553 /* ARGSUSED5 */
1554 static int
1555 tmp_symlink(
1556         struct vnode *dvp,
1557         char *lnm,
1558         struct vattr *tva,
1559         char *tnm,
1560         struct cred *cred,
1561         caller_context_t *ct,
1562         int flags)
1563 {
1564         struct tmpnode *parent = (struct tmpnode *)VTOTN(dvp);
1565         struct tmpnode *self = (struct tmpnode *)NULL;
1566         struct tmount *tm = (struct tmount *)VTOTM(dvp);
1567         char *cp = NULL;
1568         int error;
1569         size_t len;

1571         /* no symlinks allowed to files in xattr dirs */
1572         if (parent->tn_flags & ISXATTR)
1573                 return (EINVAL);

1575         error = tdirlookup(parent, lnm, &self, cred);
1576         if (error == 0) {
1577                 /*
1578                  * The entry already exists
1579                  */
```

```
1580                 tmpnode_rele(self);
1581                 return (EEXIST);        /* was 0 */
1582         }
1583
1584         if (error != ENOENT) {
1585                 if (self != NULL)
1586                         tmpnode_rele(self);
1587                 return (error);
1588         }
1589
1590         rw_enter(&parent->tn_rwlock, RW_WRITER);
1591         error = tdirenter(tm, parent, lnm, DE_CREATE, (struct tmpnode *)NULL,
1592             (struct tmpnode *)NULL, tva, &self, cred, ct);
1593         rw_exit(&parent->tn_rwlock);
1594
1595         if (error) {
1596                 if (self)
1597                         tmpnode_rele(self);
1598                 return (error);
1599         }
1600         len = strlen(tnm) + 1;
1601         cp = tmp_memalloc(len, 0);
1602         if (cp == NULL) {
1603                 tmpnode_rele(self);
1604                 return (ENOSPC);
1605         }
1606         (void) strcpy(cp, tnm);
1607
1608         self->tn_symlink = cp;
1609         self->tn_size = len - 1;
1610         tmpnode_rele(self);
1611         return (error);
1612 }
1613
1614 /* ARGSUSED2 */
1615 static int
1616 tmp_readlink(
1617         struct vnode *vp,
1618         struct uio *uiop,
1619         struct cred *cred,
1620         caller_context_t *ct)
1621 {
1622         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
1623         int error = 0;
1624
1625         if (vp->v_type != VLNK)
1626                 return (EINVAL);
1627
1628         rw_enter(&tp->tn_rwlock, RW_READER);
1629         rw_enter(&tp->tn_contents, RW_READER);
1630         error = uiomove(tp->tn_symlink, tp->tn_size, UIO_READ, uiop);
1631         gethrestime(&tp->tn_atime);
1632         rw_exit(&tp->tn_contents);
1633         rw_exit(&tp->tn_rwlock);
1634         return (error);
1635 }
1636
1637 /* ARGSUSED */
1638 static int
1639 tmp_fsync(
1640         struct vnode *vp,
1641         int syncflag,
1642         struct cred *cred,
1643         caller_context_t *ct)
1644 {
1645         return (0);
```



```
1646 }
1647
1648 /* ARGSUSED */
1649 static void
1650 tmp_inactive(struct vnode *vp, struct cred *cred, caller_context_t *ct)
1651 {
1652         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
1653         struct tmount *tm = (struct tmount *)VFSTOTM(vp->v_vfsp);
1654
1655         rw_enter(&tp->tn_rwlock, RW_WRITER);
1656 top:
1657         mutex_enter(&tp->tn_tlock);
1658         mutex_enter(&vp->v_lock);
1659         ASSERT(vp->v_count >= 1);
1660
1661         /*
1662          * If we don't have the last hold or the link count is non-zero,
1663          * there's little to do -- just drop our hold.
1664          */
1665         if (vp->v_count > 1 || tp->tn_nlink != 0) {
1666                 vp->v_count--;
1667                 mutex_exit(&vp->v_lock);
1668                 mutex_exit(&tp->tn_tlock);
1669                 rw_exit(&tp->tn_rwlock);
1670                 return;
1671         }
1672
1673         /*
1674          * We have the last hold *and* the link count is zero, so this
1675          * tmpnode is dead from the filesystem's viewpoint.  However,
1676          * if the tmpnode has any pages associated with it (i.e. if it's
1677          * a normal file with non-zero size), the tmpnode can still be
1678          * discovered by pageout or fsflush via the page vnode pointers.
1679          * In this case we must drop all our locks, truncate the tmpnode,
1680          * and try the whole dance again.
1681          */
1682         if (tp->tn_size != 0) {
1683                 if (tp->tn_type == VREG) {
1684                         mutex_exit(&vp->v_lock);
1685                         mutex_exit(&tp->tn_tlock);
1686                         rw_enter(&tp->tn_contents, RW_WRITER);
1687                         (void) tmpnode_trunc(tm, tp, 0);
1688                         rw_exit(&tp->tn_contents);
1689                         ASSERT(tp->tn_size == 0);
1690                         ASSERT(tp->tn_nblocks == 0);
1691                         goto top;
1692                 }
1693                 if (tp->tn_type == VLNK)
1694                         tmp_memfree(tp->tn_symlink, tp->tn_size + 1);
1695         }
1696
1697         /*
1698          * Remove normal file/dir's xattr dir and xattrs.
1699          */
1700         if (tp->tn_xattrdp) {
1701                 struct tmpnode *xtp = tp->tn_xattrdp;
1702
1703                 ASSERT(xtp->tn_flags & ISXATTR);
1704                 tmpnode_hold(xtp);
1705                 rw_enter(&xtp->tn_rwlock, RW_WRITER);
1706                 tdirtrunc(xtp);
1707                 DECR_COUNT(&xtp->tn_nlink, &xtp->tn_tlock);
1708                 tp->tn_xattrdp = NULL;
1709                 rw_exit(&xtp->tn_rwlock);
1710                 tmpnode_rele(xtp);
1711         }
```

```
1713         mutex_exit(&vp->v_lock);
1714         mutex_exit(&tp->tn_tlock);
1715         /* Here's our chance to send invalid event while we're between locks */
1716         vn_invalid(TNTOV(tp));
1717         mutex_enter(&tm->tm_contents);
1718         if (tp->tn_forw == NULL)
1719                 tm->tm_rootnode->tn_back = tp->tn_back;
1720         else
1721                 tp->tn_forw->tn_back = tp->tn_back;
1722         tp->tn_back->tn_forw = tp->tn_forw;
1723         mutex_exit(&tm->tm_contents);
1724         rw_exit(&tp->tn_rwlock);
1725         rw_destroy(&tp->tn_rwlock);
1726         mutex_destroy(&tp->tn_tlock);
1727         vn_free(TNTOV(tp));
1728         tmp_memfree(tp, sizeof (struct tmpnode));
1729 }

1731 /* ARGSUSED2 */
1732 static int
1733 tmp_fid(struct vnode *vp, struct fid *fidp, caller_context_t *ct)
1734 {
1735         struct tmpnode *tp = (struct tmpnode *)VTOTN(vp);
1736         struct tfid *tfid;

1738         if (fidp->fid_len < (sizeof (struct tfid) - sizeof (ushort_t))) {
1739                 fidp->fid_len = sizeof (struct tfid) - sizeof (ushort_t);
1740                 return (ENOSPC);
1741         }

1743         tfid = (struct tfid *)fidp;
1744         bzero(tfid, sizeof (struct tfid));
1745         tfid->tfid_len = (int)sizeof (struct tfid) - sizeof (ushort_t);

1747         tfid->tfid_ino = tp->tn_nodeid;
1748         tfid->tfid_gen = tp->tn_gen;

1750         return (0);
1751 }


1754 /*
1755  * Return all the pages from [off..off+len] in given file
1756  */
1757 /* ARGSUSED */
1758 static int
1759 tmp_getpage(
1760         struct vnode *vp,
1761         offset_t off,
1762         size_t len,
1763         uint_t *protp,
1764         page_t *pl[],
1765         size_t plsz,
1766         struct seg *seg,
1767         caddr_t addr,
1768         enum seg_rw rw,
1769         struct cred *cr,
1770         caller_context_t *ct)
1771 {
1772         int err = 0;
1773         struct tmpnode *tp = VTOTN(vp);
1774         anoff_t toff = (anoff_t)off;
1775         size_t tlen = len;
1776         u_offset_t tmpoff;
1777         timestruc_t now;
```



```
1779         rw_enter(&tp->tn_contents, RW_READER);

1781         if (off + len  > tp->tn_size + PAGEOFFSET) {
1782                 err = EFAULT;
1783                 goto out;
1784         }
1785         /*
1786          * Look for holes (no anon slot) in faulting range. If there are
1787          * holes we have to switch to a write lock and fill them in. Swap
1788          * space for holes was already reserved when the file was grown.
1789          */
1790         tmpoff = toff;
1791         if (non_anon(tp->tn_anon, btop(off), &tmpoff, &tlen)) {
1792                 if (!rw_tryupgrade(&tp->tn_contents)) {
1793                         rw_exit(&tp->tn_contents);
1794                         rw_enter(&tp->tn_contents, RW_WRITER);
1795                         /* Size may have changed when lock was dropped */
1796                         if (off + len  > tp->tn_size + PAGEOFFSET) {
1797                                 err = EFAULT;
1798                                 goto out;
1799                         }
1800                 }
1801                 for (toff = (anoff_t)off; toff < (anoff_t)off + len;
1802                     toff += PAGESIZE) {
1803                         if (anon_get_ptr(tp->tn_anon, btop(toff)) == NULL) {
1804                                 /* XXX - may allocate mem w. write lock held */
1805                                 (void) anon_set_ptr(tp->tn_anon, btop(toff),
1806                                     anon_alloc(vp, toff), ANON_SLEEP);
1807                                 tp->tn_nblocks++;
1808                         }
1809                 }
1810                 rw_downgrade(&tp->tn_contents);
1811         }


1814         err = pvn_getpages(tmp_getapage, vp, (u_offset_t)off, len, protp,
1815             pl, plsz, seg, addr, rw, cr);
  29         if (len <= PAGESIZE)
  30                 err = tmp_getapage(vp, (u_offset_t)off, len, protp, pl, plsz,
  31                     seg, addr, rw, cr);
  32         else
  33                 err = pvn_getpages(tmp_getapage, vp, (u_offset_t)off, len,
  34                     protp, pl, plsz, seg, addr, rw, cr);

1817         gethrestime(&now);
1818         tp->tn_atime = now;
1819         if (rw == S_WRITE)
1820                 tp->tn_mtime = now;

1822 out:
1823         rw_exit(&tp->tn_contents);
1824         return (err);
1825 }

1827 /*
1828  * Called from pvn_getpages to get a particular page.
  47  * Called from pvn_getpages or swap_getpage to get a particular page.
1829  */
1830 /*ARGSUSED*/
1831 static int
1832 tmp_getapage(
1833         struct vnode *vp,
1834         u_offset_t off,
1835         size_t len,
1836         uint_t *protp,
```

```
1837         page_t *pl[],
1838         size_t plsz,
1839         struct seg *seg,
1840         caddr_t addr,
1841         enum seg_rw rw,
1842         struct cred *cr)
1843 {
1844         struct page *pp;
1845         int flags;
1846         int err = 0;
1847         struct vnode *pvp;
1848         u_offset_t poff;

1850         if (protp != NULL)
1851                 *protp = PROT_ALL;
1852 again:
1853         if (pp = page_lookup(vp, off, rw == S_CREATE ? SE_EXCL : SE_SHARED)) {
1854                 if (pl) {
1855                         pl[0] = pp;
1856                         pl[1] = NULL;
1857                 } else {
1858                         page_unlock(pp);
1859                 }
1860         } else {
1861                 pp = page_create_va(vp, off, PAGESIZE,
1862                     PG_WAIT | PG_EXCL, seg, addr);
1863                 /*
1864                  * Someone raced in and created the page after we did the
1865                  * lookup but before we did the create, so go back and
1866                  * try to look it up again.
1867                  */
1868                 if (pp == NULL)
1869                         goto again;
1870                 /*
1871                  * Fill page from backing store, if any. If none, then
1872                  * either this is a newly filled hole or page must have
1873                  * been unmodified and freed so just zero it out.
1874                  */
1875                 err = swap_getphysname(vp, off, &pvp, &poff);
1876                 if (err) {
1877                         panic("tmp_getapage: no anon slot vp %p "
1878                             "off %llx pp %p\n", (void *)vp, off, (void *)pp);
1879                 }
1880                 if (pvp) {
1881                         flags = (pl == NULL ? B_ASYNC|B_READ : B_READ);
1882                         err = VOP_PAGEIO(pvp, pp, (u_offset_t)poff, PAGESIZE,
1883                             flags, cr, NULL);
1884                         if (flags & B_ASYNC)
1885                                 pp = NULL;
1886                 } else if (rw != S_CREATE) {
1887                         pagezero(pp, 0, PAGESIZE);
1888                 }
1889                 if (err && pp)
1890                         pvn_read_done(pp, B_ERROR);
1891                 if (err == 0) {
1892                         if (pl)
1893                                 pvn_plist_init(pp, pl, plsz, off, PAGESIZE, rw);
1894                         else
1895                                 pvn_io_done(pp);
1896                 }
1897         }
1898         return (err);
1899 }
```

_____***unchanged_portion_omitted_***