```
**********************************************************
   48230 Mon May 12 10:06:21 2014
new/usr/src/uts/common/fs/nfs/nfs4_rnode.c
4827 nfs4: slow file locking
4837 NFSv4 client lock retry delay upper limit should be shorter
**********************************************************
_____unchanged_portion_omitted_

/*
 * Return the current lease time for the server associated with the given
 * file.  Note that the lease time could change immediately after this
 * call.
 */

time_t
r2lease_time(rnode4_t *rp)
{
        nfs4_server_t   *sp;
        time_t          lease_time;
        mntinfo4_t      *mi = VTOMI4(RTOV4(rp));

        (void) nfs_rw_enter_sig(&mi->mi_recovlock, RW_READER, 0);

        /* this locks down sp if it is found */
        sp = find_nfs4_server(VTOMI4(RTOV4(rp)));

        if (VTOMI4(RTOV4(rp))->mi_vfsp->vfs_flag & VFS_UNMOUNTED) {
                if (sp != NULL) {
                        mutex_exit(&sp->s_lock);
                        nfs4_server_rele(sp);
                }
                nfs_rw_exit(&mi->mi_recovlock);
                return (1);             /* 1 second */
        }

        ASSERT(sp != NULL);

        lease_time = sp->s_lease_time;

        mutex_exit(&sp->s_lock);
        nfs4_server_rele(sp);
        nfs_rw_exit(&mi->mi_recovlock);

        return (lease_time);
}

/*
 * Return a list with information about all the known open instances for
 * a filesystem. The caller must call r4releopenlist() when done with the
 * list.
 *
 * We are safe at looking at os_valid and os_pending_close across dropping
 * the 'os_sync_lock' to count up the number of open streams and then
 * allocate memory for the osp list due to:
 *      -Looking at os_pending_close is safe since this routine is
 *      only called via recovery, and os_pending_close can only be set via
 *      a non-recovery operation (which are all blocked when recovery
 *      is active).
 *
 *      -Examining os_valid is safe since non-recovery operations, which
 *      could potentially switch os_valid to 0, are blocked (via
 *      nfs4_start_fop) and recovery is single-threaded per mntinfo4_t
 *      (which means we are the only recovery thread potentially acting
 *      on this open stream).
 */
```



```
nfs4_opinst_t *
r4mkopenlist(mntinfo4_t *mi)
{
        nfs4_opinst_t *reopenlist, *rep;
        rnode4_t *rp;
        vnode_t *vp;
        vfs_t *vfsp = mi->mi_vfsp;
        int numosp;
        nfs4_open_stream_t *osp;
        int index;
        open_delegation_type4 dtype;
        int hold_vnode;

        reopenlist = NULL;

        for (index = 0; index < rtable4size; index++) {
                rw_enter(&rtable4[index].r_lock, RW_READER);
                for (rp = rtable4[index].r_hashf;
                    rp != (rnode4_t *)(&rtable4[index]);
                    rp = rp->r_hashf) {

                        vp = RTOV4(rp);
                        if (vp->v_vfsp != vfsp)
                                continue;
                        hold_vnode = 0;

                        mutex_enter(&rp->r_os_lock);

                        /* Count the number of valid open_streams of the file */
                        numosp = 0;
                        for (osp = list_head(&rp->r_open_streams); osp != NULL;
                            osp = list_next(&rp->r_open_streams, osp)) {
                                mutex_enter(&osp->os_sync_lock);
                                if (osp->os_valid && !osp->os_pending_close)
                                        numosp++;
                                mutex_exit(&osp->os_sync_lock);
                        }

                        /* Fill in the valid open streams per vp */
                        if (numosp > 0) {
                                int j;

                                hold_vnode = 1;

                                /*
                                 * Add a new open instance to the list
                                 */
                                rep = kmem_zalloc(sizeof (*reopenlist),
                                    KM_SLEEP);
                                rep->re_next = reopenlist;
                                reopenlist = rep;

                                rep->re_vp = vp;
                                rep->re_osp = kmem_zalloc(
                                    numosp * sizeof (*(rep->re_osp)),
                                    KM_SLEEP);
                                rep->re_numosp = numosp;

                                j = 0;
                                for (osp = list_head(&rp->r_open_streams);
                                    osp != NULL;
                                    osp = list_next(&rp->r_open_streams, osp)) {

                                        mutex_enter(&osp->os_sync_lock);
                                        if (osp->os_valid &&
                                            !osp->os_pending_close) {
```

```
1700                                            osp->os_ref_count++;
1701                                            rep->re_osp[j] = osp;
1702                                            j++;
1703                                    }
1704                                    mutex_exit(&osp->os_sync_lock);
1705                            }
1706                            /*
1707                             * Assuming valid osp(s) stays valid between
1708                             * the time obtaining j and numosp.
1709                             */
1710                            ASSERT(j == numosp);
1711                    }

1713                    mutex_exit(&rp->r_os_lock);
1714                    /* do this here to keep v_lock > r_os_lock */
1715                    if (hold_vnode)
1716                            VN_HOLD(vp);
1717                    mutex_enter(&rp->r_statev4_lock);
1718                    if (rp->r_deleg_type != OPEN_DELEGATE_NONE) {
1719                            /*
1720                             * If this rnode holds a delegation,
1721                             * but if there are no valid open streams,
1722                             * then just discard the delegation
1723                             * without doing delegreturn.
1724                             */
1725                            if (numosp > 0)
1726                                    rp->r_deleg_needs_recovery =
1727                                        rp->r_deleg_type;
1728                    }
1729                    /* Save the delegation type for use outside the lock */
1730                    dtype = rp->r_deleg_type;
1731                    mutex_exit(&rp->r_statev4_lock);

1733                    /*
1734                     * If we have a delegation then get rid of it.
1735                     * We've set rp->r_deleg_needs_recovery so we have
1736                     * enough information to recover.
1737                     */
1738                    if (dtype != OPEN_DELEGATE_NONE) {
1739                            (void) nfs4delegreturn(rp, NFS4_DR_DISCARD);
1740                    }
1741            }
1742            rw_exit(&rtable4[index].r_lock);
1743        }
1744        return (reopenlist);
1745 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  430707 Mon May 12 10:06:21 2014
new/usr/src/uts/common/fs/nfs/nfs4_vnops.c
4827 nfs4: slow file locking
4837 NFSv4 client lock retry delay upper limit should be shorter
**********************************************************
/*
 * 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 2012 Nexenta Systems, Inc. All rights reserved.
 */

/*
 *      Copyright 1983,1984,1985,1986,1987,1988,1989 AT&T.
 *      All Rights Reserved
 */

/*
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 */

/*
 * Copyright (c) 2014, STRATO AG. 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/policy.h>
#include <sys/sdt.h>
#include <sys/list.h>
#include <sys/stat.h>
#include <sys/zone.h>

#include <rpc/types.h>
#include <rpc/auth.h>
#include <rpc/clnt.h>

#include <nfs/nfs.h>
#include <nfs/nfs_clnt.h>
#include <nfs/nfs_acl.h>
#include <nfs/lm.h>
#include <nfs/nfs4.h>
#include <nfs/nfs4_kprot.h>
#include <nfs/rnode4.h>
#include <nfs/nfs4_clnt.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>
#include <sys/int_fmtio.h>
#include <sys/fs/autofs.h>

typedef struct {
        nfs4_ga_res_t   *di_garp;
        cred_t          *di_cred;
        hrtime_t        di_time_call;
} dirattr_info_t;

typedef enum nfs4_acl_op {
        NFS4_ACL_GET,
        NFS4_ACL_SET
} nfs4_acl_op_t;

static struct lm_sysid *nfs4_find_sysid(mntinfo4_t *mi);

static void     nfs4_update_dircaches(change_info4 *, vnode_t *, vnode_t *,
                        char *, dirattr_info_t *);

static void     nfs4close_otw(rnode4_t *, cred_t *, nfs4_open_owner_t *,
                    nfs4_open_stream_t *, int *, int *, nfs4_close_type_t,
                    nfs4_error_t *, int *);
static int      nfs4_rdwrlbn(vnode_t *, page_t *, u_offset_t, size_t, int,
```

```
127                         cred_t *);
128 static int      nfs4write(vnode_t *, caddr_t, u_offset_t, int, cred_t *,
129                         stable_how4 *);
130 static int      nfs4read(vnode_t *, caddr_t, offset_t, int, size_t *,
131                         cred_t *, bool_t, struct uio *);
132 static int      nfs4setattr(vnode_t *, struct vattr *, int, cred_t *,
133                         vsecattr_t *);
134 static int      nfs4openattr(vnode_t *, vnode_t **, int, cred_t *);
135 static int      nfs4lookup(vnode_t *, char *, vnode_t **, cred_t *, int);
136 static int      nfs4lookup_xattr(vnode_t *, char *, vnode_t **, int, cred_t *);
137 static int      nfs4lookupvalidate_otw(vnode_t *, char *, vnode_t **, cred_t *);
138 static int      nfs4lookupnew_otw(vnode_t *, char *, vnode_t **, cred_t *);
139 static int      nfs4mknod(vnode_t *, char *, struct vattr *, enum vcexcl,
140                         int, vnode_t **, cred_t *);
141 static int      nfs4open_otw(vnode_t *, char *, struct vattr *, vnode_t **,
142                         cred_t *, int, int, enum createmode4, int);
143 static int      nfs4rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
144                         caller_context_t *);
145 static int      nfs4rename_persistent_fh(vnode_t *, char *, vnode_t *,
146                         vnode_t *, char *, cred_t *, nfsstat4 *);
147 static int      nfs4rename_volatile_fh(vnode_t *, char *, vnode_t *,
148                         vnode_t *, char *, cred_t *, nfsstat4 *);
149 static int      do_nfs4readdir(vnode_t *, rddir4_cache *, cred_t *);
150 static void     nfs4readdir(vnode_t *, rddir4_cache *, cred_t *);
151 static int      nfs4_bio(struct buf *, stable_how4 *, cred_t *, bool_t);
152 static int      nfs4_getapage(vnode_t *, u_offset_t, size_t, uint_t *,
153                         page_t *[], size_t, struct seg *, caddr_t,
154                         enum seg_rw, cred_t *);
155 static void     nfs4_readahead(vnode_t *, u_offset_t, caddr_t, struct seg *,
156                         cred_t *);
157 static int      nfs4_sync_putapage(vnode_t *, page_t *, u_offset_t, size_t,
158                         int, cred_t *);
159 static int      nfs4_sync_pageio(vnode_t *, page_t *, u_offset_t, size_t,
160                         int, cred_t *);
161 static int      nfs4_commit(vnode_t *, offset4, count4, cred_t *);
162 static void     nfs4_set_mod(vnode_t *);
163 static void     nfs4_get_commit(vnode_t *);
164 static void     nfs4_get_commit_range(vnode_t *, u_offset_t, size_t);
165 static int      nfs4_putpage_commit(vnode_t *, offset_t, size_t, cred_t *);
166 static int      nfs4_commit_vp(vnode_t *, u_offset_t, size_t, cred_t *, int);
167 static int      nfs4_sync_commit(vnode_t *, page_t *, offset3, count3,
168                         cred_t *);
169 static void     do_nfs4_async_commit(vnode_t *, page_t *, offset3, count3,
170                         cred_t *);
171 static int      nfs4_update_attrcache(nfsstat4, nfs4_ga_res_t *,
172                         hrtime_t, vnode_t *, cred_t *);
173 static int      nfs4_open_non_reg_file(vnode_t **, int, cred_t *);
174 static int      nfs4_safelock(vnode_t *, const struct flock64 *, cred_t *);
175 static void     nfs4_register_lock_locally(vnode_t *, struct flock64 *, int,
176                         u_offset_t);
177 static int      nfs4_lockrelease(vnode_t *, int, offset_t, cred_t *);
178 static int      nfs4_block_and_wait(clock_t *, rnode4_t *);
179 static cred_t  *state_to_cred(nfs4_open_stream_t *);
180 static void     denied_to_flk(LOCK4denied *, flock64_t *, LOCKT4args *);
181 static pid_t    lo_to_pid(lock_owner4 *);
182 static void     nfs4_reinstitute_local_lock_state(vnode_t *, flock64_t *,
183                         cred_t *, nfs4_lock_owner_t *);
184 static void     push_reinstate(vnode_t *, int, flock64_t *, cred_t *,
185                         nfs4_lock_owner_t *);
186 static int      open_and_get_osp(vnode_t *, cred_t *, nfs4_open_stream_t **);
187 static void     nfs4_delmap_callback(struct as *, void *, uint_t);
188 static void     nfs4_free_delmapcall(nfs4_delmapcall_t *);
189 static nfs4_delmapcall_t        *nfs4_init_delmapcall();
190 static int      nfs4_find_and_delete_delmapcall(rnode4_t *, int *);
191 static int      nfs4_is_acl_mask_valid(uint_t, nfs4_acl_op_t);
192 static int      nfs4_create_getsecattr_return(vsecattr_t *, vsecattr_t *,
```



```
193                         uid_t, gid_t, int);
194
195 /*
196  * Routines that implement the setting of v4 args for the misc. ops
197  */
198 static void     nfs4args_lock_free(nfs_argop4 *);
199 static void     nfs4args_lockt_free(nfs_argop4 *);
200 static void     nfs4args_setattr(nfs_argop4 *, vattr_t *, vsecattr_t *,
201                         int, rnode4_t *, cred_t *, bitmap4, int *,
202                         nfs4_stateid_types_t *);
203 static void     nfs4args_setattr_free(nfs_argop4 *);
204 static int      nfs4args_verify(nfs_argop4 *, vattr_t *, enum nfs_opnum4,
205                         bitmap4);
206 static void     nfs4args_verify_free(nfs_argop4 *);
207 static void     nfs4args_write(nfs_argop4 *, stable_how4, rnode4_t *, cred_t *,
208                         WRITE4args **, nfs4_stateid_types_t *);
209
210 /*
211  * These are the vnode ops functions that implement the vnode interface to
212  * the networked file system.  See more comments below at nfs4_vnodeops.
213  */
214 static int      nfs4_open(vnode_t **, int, cred_t *, caller_context_t *);
215 static int      nfs4_close(vnode_t *, int, int, offset_t, cred_t *,
216                         caller_context_t *);
217 static int      nfs4_read(vnode_t *, struct uio *, int, cred_t *,
218                         caller_context_t *);
219 static int      nfs4_write(vnode_t *, struct uio *, int, cred_t *,
220                         caller_context_t *);
221 static int      nfs4_ioctl(vnode_t *, int, intptr_t, int, cred_t *, int *,
222                         caller_context_t *);
223 static int      nfs4_setattr(vnode_t *, struct vattr *, int, cred_t *,
224                         caller_context_t *);
225 static int      nfs4_access(vnode_t *, int, int, cred_t *, caller_context_t *);
226 static int      nfs4_readlink(vnode_t *, struct uio *, cred_t *,
227                         caller_context_t *);
228 static int      nfs4_fsync(vnode_t *, int, cred_t *, caller_context_t *);
229 static int      nfs4_create(vnode_t *, char *, struct vattr *, enum vcexcl,
230                         int, vnode_t **, cred_t *, int, caller_context_t *,
231                         vsecattr_t *);
232 static int      nfs4_remove(vnode_t *, char *, cred_t *, caller_context_t *,
233                         int);
234 static int      nfs4_link(vnode_t *, vnode_t *, char *, cred_t *,
235                         caller_context_t *, int);
236 static int      nfs4_rename(vnode_t *, char *, vnode_t *, char *, cred_t *,
237                         caller_context_t *, int);
238 static int      nfs4_mkdir(vnode_t *, char *, struct vattr *, vnode_t **,
239                         cred_t *, caller_context_t *, int, vsecattr_t *);
240 static int      nfs4_rmdir(vnode_t *, char *, vnode_t *, cred_t *,
241                         caller_context_t *, int);
242 static int      nfs4_symlink(vnode_t *, char *, struct vattr *, char *,
243                         cred_t *, caller_context_t *, int);
244 static int      nfs4_readdir(vnode_t *, struct uio *, cred_t *, int *,
245                         caller_context_t *, int);
246 static int      nfs4_seek(vnode_t *, offset_t, offset_t *, caller_context_t *);
247 static int      nfs4_getpage(vnode_t *, offset_t, size_t, uint_t *,
248                         page_t *[], size_t, struct seg *, caddr_t,
249                         enum seg_rw, cred_t *, caller_context_t *);
250 static int      nfs4_putpage(vnode_t *, offset_t, size_t, int, cred_t *,
251                         caller_context_t *);
252 static int      nfs4_map(vnode_t *, offset_t, struct as *, caddr_t *, size_t,
253                         uchar_t, uchar_t, uint_t, cred_t *, caller_context_t *);
254 static int      nfs4_addmap(vnode_t *, offset_t, struct as *, caddr_t, size_t,
255                         uchar_t, uchar_t, uint_t, cred_t *, caller_context_t *);
256 static int      nfs4_cmp(vnode_t *, vnode_t *, caller_context_t *);
257 static int      nfs4_frlock(vnode_t *, int, struct flock64 *, int, offset_t,
258                         struct flk_callback *, cred_t *, caller_context_t *);
```

```
259 static int      nfs4_space(vnode_t *, int, struct flock64 *, int, offset_t,
260                         cred_t *, caller_context_t *);
261 static int      nfs4_delmap(vnode_t *, offset_t, struct as *, caddr_t, size_t,
262                         uint_t, uint_t, uint_t, cred_t *, caller_context_t *);
263 static int      nfs4_pageio(vnode_t *, page_t *, u_offset_t, size_t, int,
264                         cred_t *, caller_context_t *);
265 static void     nfs4_dispose(vnode_t *, page_t *, int, int, cred_t *,
266                         caller_context_t *);
267 static int      nfs4_setsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
268                         caller_context_t *);
269 /*
270  * These vnode ops are required to be called from outside this source file,
271  * e.g. by ephemeral mount stub vnode ops, and so may not be declared
272  * as static.
273  */
274 int     nfs4_getattr(vnode_t *, struct vattr *, int, cred_t *,
275             caller_context_t *);
276 void    nfs4_inactive(vnode_t *, cred_t *, caller_context_t *);
277 int     nfs4_lookup(vnode_t *, char *, vnode_t **,
278             struct pathname *, int, vnode_t *, cred_t *,
279             caller_context_t *, int *, pathname_t *);
280 int     nfs4_fid(vnode_t *, fid_t *, caller_context_t *);
281 int     nfs4_rwlock(vnode_t *, int, caller_context_t *);
282 void    nfs4_rwunlock(vnode_t *, int, caller_context_t *);
283 int     nfs4_realvp(vnode_t *, vnode_t **, caller_context_t *);
284 int     nfs4_pathconf(vnode_t *, int, ulong_t *, cred_t *,
285             caller_context_t *);
286 int     nfs4_getsecattr(vnode_t *, vsecattr_t *, int, cred_t *,
287             caller_context_t *);
288 int     nfs4_shrlock(vnode_t *, int, struct shrlock *, int, cred_t *,
289             caller_context_t *);

291 /*
292  * Used for nfs4_commit_vp() to indicate if we should
293  * wait on pending writes.
294  */
295 #define NFS4_WRITE_NOWAIT       0
296 #define NFS4_WRITE_WAIT         1

 38 #define NFS4_BASE_WAIT_TIME 1   /* 1 second */

298 /*
299  * Error flags used to pass information about certain special errors
300  * which need to be handled specially.
301  */
302 #define NFS_EOF                 -98
303 #define NFS_VERF_MISMATCH       -97

305 /*
306  * Flags used to differentiate between which operation drove the
307  * potential CLOSE OTW. (see nfs4_close_otw_if_necessary)
308  */
309 #define NFS4_CLOSE_OP           0x1
310 #define NFS4_DELMAP_OP          0x2
311 #define NFS4_INACTIVE_OP        0x3

313 #define ISVDEV(t) ((t == VBLK) || (t == VCHR) || (t == VFIFO))

315 /* ALIGN64 aligns the given buffer and adjust buffer size to 64 bit */
316 #define ALIGN64(x, ptr, sz)                                             \
317         x = ((uintptr_t)(ptr)) & (sizeof (uint64_t) - 1);               \
318         if (x) {                                                        \
319                 x = sizeof (uint64_t) - (x);                            \
320                 sz -= (x);                                              \
321                 ptr += (x);                                             \
322         }
```



```
324 #ifdef DEBUG
325 int nfs4_client_attr_debug = 0;
326 int nfs4_client_state_debug = 0;
327 int nfs4_client_shadow_debug = 0;
328 int nfs4_client_lock_debug = 0;
329 int nfs4_seqid_sync = 0;
330 int nfs4_client_map_debug = 0;
331 static int nfs4_pageio_debug = 0;
332 int nfs4_client_inactive_debug = 0;
333 int nfs4_client_recov_debug = 0;
334 int nfs4_client_failover_debug = 0;
335 int nfs4_client_call_debug = 0;
336 int nfs4_client_lookup_debug = 0;
337 int nfs4_client_zone_debug = 0;
338 int nfs4_lost_rqst_debug = 0;
339 int nfs4_rdattrerr_debug = 0;
340 int nfs4_open_stream_debug = 0;

342 int nfs4read_error_inject;

344 static int nfs4_create_misses = 0;

346 static int nfs4_readdir_cache_shorts = 0;
347 static int nfs4_readdir_readahead = 0;

349 static int nfs4_bio_do_stop = 0;

351 static int nfs4_lostpage = 0;   /* number of times we lost original page */

353 int nfs4_mmap_debug = 0;

355 static int nfs4_pathconf_cache_hits = 0;
356 static int nfs4_pathconf_cache_misses = 0;

358 int nfs4close_all_cnt;
359 int nfs4close_one_debug = 0;
360 int nfs4close_notw_debug = 0;

362 int denied_to_flk_debug = 0;
363 void *lockt_denied_debug;

365 #endif

367 /*
368  * In milliseconds. Should be less than half of the lease time or better,
369  * less than one second.
370  */
371 int nfs4_base_wait_time = 20;

373 /*
374 #endif /* ! codereview */
375  * How long to wait before trying again if OPEN_CONFIRM gets ETIMEDOUT
376  * or NFS4ERR_RESOURCE.
377  */
378 static int confirm_retry_sec = 30;

380 static int nfs4_lookup_neg_cache = 1;

382 /*
383  * number of pages to read ahead
384  * optimized for 100 base-T.
385  */
386 static int nfs4_nra = 4;

388 static int nfs4_do_symlink_cache = 1;
```

```
390 static int nfs4_pathconf_disable_cache = 0;

392 /*
393  * These are the vnode ops routines which implement the vnode interface to
394  * the networked file system.  These routines just take their parameters,
395  * make them look networkish by putting the right info into interface structs,
396  * and then calling the appropriate remote routine(s) to do the work.
397  *
398  * Note on directory name lookup cacheing:  If we detect a stale fhandle,
399  * we purge the directory cache relative to that vnode.  This way, the
400  * user won't get burned by the cache repeatedly.  See <nfs/rnode4.h> for
401  * more details on rnode locking.
402  */

404 struct vnodeops *nfs4_vnodeops;

406 const fs_operation_def_t nfs4_vnodeops_template[] = {
407         VOPNAME_OPEN,           { .vop_open = nfs4_open },
408         VOPNAME_CLOSE,          { .vop_close = nfs4_close },
409         VOPNAME_READ,           { .vop_read = nfs4_read },
410         VOPNAME_WRITE,          { .vop_write = nfs4_write },
411         VOPNAME_IOCTL,          { .vop_ioctl = nfs4_ioctl },
412         VOPNAME_GETATTR,        { .vop_getattr = nfs4_getattr },
413         VOPNAME_SETATTR,        { .vop_setattr = nfs4_setattr },
414         VOPNAME_ACCESS,         { .vop_access = nfs4_access },
415         VOPNAME_LOOKUP,         { .vop_lookup = nfs4_lookup },
416         VOPNAME_CREATE,         { .vop_create = nfs4_create },
417         VOPNAME_REMOVE,         { .vop_remove = nfs4_remove },
418         VOPNAME_LINK,           { .vop_link = nfs4_link },
419         VOPNAME_RENAME,         { .vop_rename = nfs4_rename },
420         VOPNAME_MKDIR,          { .vop_mkdir = nfs4_mkdir },
421         VOPNAME_RMDIR,          { .vop_rmdir = nfs4_rmdir },
422         VOPNAME_READDIR,        { .vop_readdir = nfs4_readdir },
423         VOPNAME_SYMLINK,        { .vop_symlink = nfs4_symlink },
424         VOPNAME_READLINK,       { .vop_readlink = nfs4_readlink },
425         VOPNAME_FSYNC,          { .vop_fsync = nfs4_fsync },
426         VOPNAME_INACTIVE,       { .vop_inactive = nfs4_inactive },
427         VOPNAME_FID,            { .vop_fid = nfs4_fid },
428         VOPNAME_RWLOCK,         { .vop_rwlock = nfs4_rwlock },
429         VOPNAME_RWUNLOCK,       { .vop_rwunlock = nfs4_rwunlock },
430         VOPNAME_SEEK,           { .vop_seek = nfs4_seek },
431         VOPNAME_FRLOCK,         { .vop_frlock = nfs4_frlock },
432         VOPNAME_SPACE,          { .vop_space = nfs4_space },
433         VOPNAME_REALVP,         { .vop_realvp = nfs4_realvp },
434         VOPNAME_GETPAGE,        { .vop_getpage = nfs4_getpage },
435         VOPNAME_PUTPAGE,        { .vop_putpage = nfs4_putpage },
436         VOPNAME_MAP,            { .vop_map = nfs4_map },
437         VOPNAME_ADDMAP,         { .vop_addmap = nfs4_addmap },
438         VOPNAME_DELMAP,         { .vop_delmap = nfs4_delmap },
439         /* no separate nfs4_dump */
440         VOPNAME_DUMP,           { .vop_dump = nfs_dump },
441         VOPNAME_PATHCONF,       { .vop_pathconf = nfs4_pathconf },
442         VOPNAME_PAGEIO,         { .vop_pageio = nfs4_pageio },
443         VOPNAME_DISPOSE,        { .vop_dispose = nfs4_dispose },
444         VOPNAME_SETSECATTR,     { .vop_setsecattr = nfs4_setsecattr },
445         VOPNAME_GETSECATTR,     { .vop_getsecattr = nfs4_getsecattr },
446         VOPNAME_SHRLOCK,        { .vop_shrlock = nfs4_shrlock },
447         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
448         NULL,                   NULL
449 };

451 /*
452  * The following are subroutines and definitions to set args or get res
453  * for the different nfsv4 ops
454  */
```



```
456 void
457 nfs4args_lookup_free(nfs_argop4 *argop, int arglen)
458 {
459         int             i;

461         for (i = 0; i < arglen; i++) {
462                 if (argop[i].argop == OP_LOOKUP) {
463                         kmem_free(
464                             argop[i].nfs_argop4_u.oplookup.
465                             objname.utf8string_val,
466                             argop[i].nfs_argop4_u.oplookup.
467                             objname.utf8string_len);
468                 }
469         }
470 }

472 static void
473 nfs4args_lock_free(nfs_argop4 *argop)
474 {
475         locker4 *locker = &argop->nfs_argop4_u.oplock.locker;

477         if (locker->new_lock_owner == TRUE) {
478                 open_to_lock_owner4 *open_owner;

480                 open_owner = &locker->locker4_u.open_owner;
481                 if (open_owner->lock_owner.owner_val != NULL) {
482                         kmem_free(open_owner->lock_owner.owner_val,
483                             open_owner->lock_owner.owner_len);
484                 }
485         }
486 }

488 static void
489 nfs4args_lockt_free(nfs_argop4 *argop)
490 {
491         lock_owner4 *lowner = &argop->nfs_argop4_u.oplockt.owner;

493         if (lowner->owner_val != NULL) {
494                 kmem_free(lowner->owner_val, lowner->owner_len);
495         }
496 }

498 static void
499 nfs4args_setattr(nfs_argop4 *argop, vattr_t *vap, vsecattr_t *vsap, int flags,
500     rnode4_t *rp, cred_t *cr, bitmap4 supp, int *error,
501     nfs4_stateid_types_t *sid_types)
502 {
503         fattr4          *attr = &argop->nfs_argop4_u.opsetattr.obj_attributes;
504         mntinfo4_t      *mi;

506         argop->argop = OP_SETATTR;
507         /*
508          * The stateid is set to 0 if client is not modifying the size
509          * and otherwise to whatever nfs4_get_stateid() returns.
510          *
511          * XXX Note: nfs4_get_stateid() returns 0 if no lockowner and/or no
512          * state struct could be found for the process/file pair.  We may
513          * want to change this in the future (by OPENing the file).  See
514          * bug # 4474852.
515          */
516         if (vap->va_mask & AT_SIZE) {

518                 ASSERT(rp != NULL);
519                 mi = VTOMI4(RTOV4(rp));
```

```
521                 argop->nfs_argop4_u.opsetattr.stateid =
522                     nfs4_get_stateid(cr, rp, curproc->p_pidp->pid_id, mi,
523                     OP_SETATTR, sid_types, FALSE);
524         } else {
525                 bzero(&argop->nfs_argop4_u.opsetattr.stateid,
526                     sizeof (stateid4));
527         }
528
529         *error = vattr_to_fattr4(vap, vsap, attr, flags, OP_SETATTR, supp);
530         if (*error)
531                 bzero(attr, sizeof (*attr));
532 }
533
534 static void
535 nfs4args_setattr_free(nfs_argop4 *argop)
536 {
537         nfs4_fattr4_free(&argop->nfs_argop4_u.opsetattr.obj_attributes);
538 }
539
540 static int
541 nfs4args_verify(nfs_argop4 *argop, vattr_t *vap, enum nfs_opnum4 op,
542     bitmap4 supp)
543 {
544         fattr4 *attr;
545         int error = 0;
546
547         argop->argop = op;
548         switch (op) {
549         case OP_VERIFY:
550                 attr = &argop->nfs_argop4_u.opverify.obj_attributes;
551                 break;
552         case OP_NVERIFY:
553                 attr = &argop->nfs_argop4_u.opnverify.obj_attributes;
554                 break;
555         default:
556                 return (EINVAL);
557         }
558         if (!error)
559                 error = vattr_to_fattr4(vap, NULL, attr, 0, op, supp);
560         if (error)
561                 bzero(attr, sizeof (*attr));
562         return (error);
563 }
564
565 static void
566 nfs4args_verify_free(nfs_argop4 *argop)
567 {
568         switch (argop->argop) {
569         case OP_VERIFY:
570                 nfs4_fattr4_free(&argop->nfs_argop4_u.opverify.obj_attributes);
571                 break;
572         case OP_NVERIFY:
573                 nfs4_fattr4_free(&argop->nfs_argop4_u.opnverify.obj_attributes);
574                 break;
575         default:
576                 break;
577         }
578 }
579
580 static void
581 nfs4args_write(nfs_argop4 *argop, stable_how4 stable, rnode4_t *rp, cred_t *cr,
582     WRITE4args **wargs_pp, nfs4_stateid_types_t *sid_tp)
583 {
584         WRITE4args *wargs = &argop->nfs_argop4_u.opwrite;
585         mntinfo4_t *mi = VTOMI4(RTOV4(rp));
```



```
586
587         argop->argop = OP_WRITE;
588         wargs->stable = stable;
589         wargs->stateid = nfs4_get_w_stateid(cr, rp, curproc->p_pidp->pid_id,
590             mi, OP_WRITE, sid_tp);
591         wargs->mblk = NULL;
592         *wargs_pp = wargs;
593 }
594
595 void
596 nfs4args_copen_free(OPEN4cargs *open_args)
597 {
598         if (open_args->owner.owner_val) {
599                 kmem_free(open_args->owner.owner_val,
600                     open_args->owner.owner_len);
601         }
602         if ((open_args->opentype == OPEN4_CREATE) &&
603             (open_args->mode != EXCLUSIVE4)) {
604                 nfs4_fattr4_free(&open_args->createhow4_u.createattrs);
605         }
606 }
607
608 /*
609  * XXX:  This is referenced in modstubs.s
610  */
611 struct vnodeops *
612 nfs4_getvnodeops(void)
613 {
614         return (nfs4_vnodeops);
615 }
616
617 /*
618  * The OPEN operation opens a regular file.
619  */
620 /*ARGSUSED3*/
621 static int
622 nfs4_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
623 {
624         vnode_t *dvp = NULL;
625         rnode4_t *rp, *drp;
626         int error;
627         int just_been_created;
628         char fn[MAXNAMELEN];
629
630         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE, "nfs4_open: "));
631         if (nfs_zone() != VTOMI4(*vpp)->mi_zone)
632                 return (EIO);
633         rp = VTOR4(*vpp);
634
635         /*
636          * Check to see if opening something besides a regular file;
637          * if so skip the OTW call
638          */
639         if ((*vpp)->v_type != VREG) {
640                 error = nfs4_open_non_reg_file(vpp, flag, cr);
641                 return (error);
642         }
643
644         /*
645          * XXX - would like a check right here to know if the file is
646          * executable or not, so as to skip OTW
647          */
648
649         if ((error = vtodv(*vpp, &dvp, cr, TRUE)) != 0)
650                 return (error);
651
652         drp = VTOR4(dvp);
```

```
653         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR4(dvp)))
654                 return (EINTR);
655
656         if ((error = vtoname(*vpp, fn, MAXNAMELEN)) != 0) {
657                 nfs_rw_exit(&drp->r_rwlock);
658                 return (error);
659         }
660
661         /*
662          * See if this file has just been CREATEd.
663          * If so, clear the flag and update the dnlc, which was previously
664          * skipped in nfs4_create.
665          * XXX need better serilization on this.
666          * XXX move this into the nf4open_otw call, after we have
667          * XXX acquired the open owner seqid sync.
668          */
669         mutex_enter(&rp->r_statev4_lock);
670         if (rp->created_v4) {
671                 rp->created_v4 = 0;
672                 mutex_exit(&rp->r_statev4_lock);
673
674                 dnlc_update(dvp, fn, *vpp);
675                 /* This is needed so we don't bump the open ref count */
676                 just_been_created = 1;
677         } else {
678                 mutex_exit(&rp->r_statev4_lock);
679                 just_been_created = 0;
680         }
681
682         /*
683          * If caller specified O_TRUNC/FTRUNC, then be sure to set
684          * FWRITE (to drive successful setattr(size=0) after open)
685          */
686         if (flag & FTRUNC)
687                 flag |= FWRITE;
688
689         error = nfs4open_otw(dvp, fn, NULL, vpp, cr, 0, flag, 0,
690             just_been_created);
691
692         if (!error && !((*vpp)->v_flag & VROOT))
693                 dnlc_update(dvp, fn, *vpp);
694
695         nfs_rw_exit(&drp->r_rwlock);
696
697         /* release the hold from vtodv */
698         VN_RELE(dvp);
699
700         /* exchange the shadow for the master vnode, if needed */
701
702         if (error == 0 && IS_SHADOW(*vpp, rp))
703                 sv_exchange(vpp);
704
705         return (error);
706 }
707
708 /*
709  * See if there's a "lost open" request to be saved and recovered.
710  */
711 static void
712 nfs4open_save_lost_rqst(int error, nfs4_lost_rqst_t *lost_rqstp,
713     nfs4_open_owner_t *oop, cred_t *cr, vnode_t *vp,
714     vnode_t *dvp, OPEN4cargs *open_args)
715 {
716         vfs_t *vfsp;
717         char *srccfp;
```



```
719         vfsp = (dvp ? dvp->v_vfsp : vp->v_vfsp);
720
721         if (error != ETIMEDOUT && error != EINTR &&
722             !NFS4_FRC_UNMT_ERR(error, vfsp)) {
723                 lost_rqstp->lr_op = 0;
724                 return;
725         }
726
727         NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE,
728             "nfs4open_save_lost_rqst: error %d", error));
729
730         lost_rqstp->lr_op = OP_OPEN;
731
732         /*
733          * The vp (if it is not NULL) and dvp are held and rele'd via
734          * the recovery code.  See nfs4_save_lost_rqst.
735          */
736         lost_rqstp->lr_vp = vp;
737         lost_rqstp->lr_dvp = dvp;
738         lost_rqstp->lr_oop = oop;
739         lost_rqstp->lr_osp = NULL;
740         lost_rqstp->lr_lop = NULL;
741         lost_rqstp->lr_cr = cr;
742         lost_rqstp->lr_flk = NULL;
743         lost_rqstp->lr_oacc = open_args->share_access;
744         lost_rqstp->lr_odeny = open_args->share_deny;
745         lost_rqstp->lr_oclaim = open_args->claim;
746         if (open_args->claim == CLAIM_DELEGATE_CUR) {
747                 lost_rqstp->lr_ostateid =
748                     open_args->open_claim4_u.delegate_cur_info.delegate_stateid;
749                 srccfp = open_args->open_claim4_u.delegate_cur_info.cfile;
750         } else {
751                 srccfp = open_args->open_claim4_u.cfile;
752         }
753         lost_rqstp->lr_ofile.utf8string_len = 0;
754         lost_rqstp->lr_ofile.utf8string_val = NULL;
755         (void) str_to_utf8(srccfp, &lost_rqstp->lr_ofile);
756         lost_rqstp->lr_putfirst = FALSE;
757 }
758
759 struct nfs4_excl_time {
760         uint32 seconds;
761         uint32 nseconds;
762 };
763
764 /*
765  * The OPEN operation creates and/or opens a regular file
766  *
767  * ARGSUSED
768  */
769 static int
770 nfs4open_otw(vnode_t *dvp, char *file_name, struct vattr *in_va,
771     vnode_t **vpp, cred_t *cr, int create_flag, int open_flag,
772     enum createmode4 createmode, int file_just_been_created)
773 {
774         rnode4_t *rp;
775         rnode4_t *drp = VTOR4(dvp);
776         vnode_t *vp = NULL;
777         vnode_t *vpi = *vpp;
778         bool_t needrecov = FALSE;
779
780         int doqueue = 1;
781
782         COMPOUND4args_clnt args;
783         COMPOUND4res_clnt res;
784         nfs_argop4 *argop;
```

```
785         nfs_resop4 *resop;
786         int argoplist_size;
787         int idx_open, idx_fattr;

789         GETFH4res *gf_res = NULL;
790         OPEN4res *op_res = NULL;
791         nfs4_ga_res_t *garp;
792         fattr4 *attr = NULL;
793         struct nfs4_excl_time verf;
794         bool_t did_excl_setup = FALSE;
795         int created_osp;

797         OPEN4cargs *open_args;
798         nfs4_open_owner_t       *oop = NULL;
799         nfs4_open_stream_t      *osp = NULL;
800         seqid4 seqid = 0;
801         bool_t retry_open = FALSE;
802         nfs4_recov_state_t recov_state;
803         nfs4_lost_rqst_t lost_rqst;
804         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
805         hrtime_t t;
806         int acc = 0;
807         cred_t *cred_otw = NULL;        /* cred used to do the RPC call */
808         cred_t *ncr = NULL;

810         nfs4_sharedfh_t *otw_sfh;
811         nfs4_sharedfh_t *orig_sfh;
812         int fh_differs = 0;
813         int numops, setgid_flag;
814         int num_bseqid_retry = NFS4_NUM_RETRY_BAD_SEQID + 1;

816         /*
817          * Make sure we properly deal with setting the right gid on
818          * a newly created file to reflect the parent's setgid bit
819          */
820         setgid_flag = 0;
821         if (create_flag && in_va) {

823                 /*
824                  * If there is grpid mount flag used or
825                  * the parent's directory has the setgid bit set
826                  * _and_ the client was able to get a valid mapping
827                  * for the parent dir's owner_group, we want to
828                  * append NVERIFY(owner_group == dva.va_gid) and
829                  * SETATTR to the CREATE compound.
830                  */
831                 mutex_enter(&drp->r_statelock);
832                 if ((VTOMI4(dvp)->mi_flags & MI4_GRPID ||
833                     drp->r_attr.va_mode & VSGID) &&
834                     drp->r_attr.va_gid != GID_NOBODY) {
835                         in_va->va_mask |= AT_GID;
836                         in_va->va_gid = drp->r_attr.va_gid;
837                         setgid_flag = 1;
838                 }
839                 mutex_exit(&drp->r_statelock);
840         }

842         /*
843          * Normal/non-create compound:
844          * PUTFH(dfh) + OPEN(create) + GETFH + GETATTR(new)
845          *
846          * Open(create) compound no setgid:
847          * PUTFH(dfh) + SAVEFH + OPEN(create) + GETFH + GETATTR(new) +
848          * RESTOREFH + GETATTR
849          *
850          * Open(create) setgid:
```



```
851          * PUTFH(dfh) + OPEN(create) + GETFH + GETATTR(new) +
852          * SAVEFH + PUTFH(dfh) + GETATTR(dvp) + RESTOREFH +
853          * NVERIFY(grp) + SETATTR
854          */
855         if (setgid_flag) {
856                 numops = 10;
857                 idx_open = 1;
858                 idx_fattr = 3;
859         } else if (create_flag) {
860                 numops = 7;
861                 idx_open = 2;
862                 idx_fattr = 4;
863         } else {
864                 numops = 4;
865                 idx_open = 1;
866                 idx_fattr = 3;
867         }

869         args.array_len = numops;
870         argoplist_size = numops * sizeof (nfs_argop4);
871         argop = kmem_alloc(argoplist_size, KM_SLEEP);

873         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE, "nfs4open_otw: "
874             "open %s open flag 0x%x cred %p", file_name, open_flag,
875             (void *)cr));

877         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);
878         if (create_flag) {
879                 /*
880                  * We are to create a file.  Initialize the passed in vnode
881                  * pointer.
882                  */
883                 vpi = NULL;
884         } else {
885                 /*
886                  * Check to see if the client owns a read delegation and is
887                  * trying to open for write.  If so, then return the delegation
888                  * to avoid the server doing a cb_recall and returning DELAY.
889                  * NB - we don't use the statev4_lock here because we'd have
890                  * to drop the lock anyway and the result would be stale.
891                  */
892                 if ((open_flag & FWRITE) &&
893                     VTOR4(vpi)->r_deleg_type == OPEN_DELEGATE_READ)
894                         (void) nfs4delegreturn(VTOR4(vpi), NFS4_DR_REOPEN);

896                 /*
897                  * If the file has a delegation, then do an access check up
898                  * front.  This avoids having to an access check later after
899                  * we've already done start_op, which could deadlock.
900                  */
901                 if (VTOR4(vpi)->r_deleg_type != OPEN_DELEGATE_NONE) {
902                         if (open_flag & FREAD &&
903                             nfs4_access(vpi, VREAD, 0, cr, NULL) == 0)
904                                 acc |= VREAD;
905                         if (open_flag & FWRITE &&
906                             nfs4_access(vpi, VWRITE, 0, cr, NULL) == 0)
907                                 acc |= VWRITE;
908                 }
909         }

911         drp = VTOR4(dvp);

913         recov_state.rs_flags = 0;
914         recov_state.rs_num_retry_despite_err = 0;
915         cred_otw = cr;
```

```
917 recov_retry:
918         fh_differs = 0;
919         nfs4_error_zinit(&e);

921         e.error = nfs4_start_op(VTOMI4(dvp), dvp, vpi, &recov_state);
922         if (e.error) {
923                 if (ncr != NULL)
924                         crfree(ncr);
925                 kmem_free(argop, argoplist_size);
926                 return (e.error);
927         }

929         args.ctag = TAG_OPEN;
930         args.array_len = numops;
931         args.array = argop;

933         /* putfh directory fh */
934         argop[0].argop = OP_CPUTFH;
935         argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

937         /* OPEN: either op 1 or op 2 depending upon create/setgid flags */
938         argop[idx_open].argop = OP_COPEN;
939         open_args = &argop[idx_open].nfs_argop4_u.opcopen;
940         open_args->claim = CLAIM_NULL;

942         /* name of file */
943         open_args->open_claim4_u.cfile = file_name;
944         open_args->owner.owner_len = 0;
945         open_args->owner.owner_val = NULL;

947         if (create_flag) {
948                 /* CREATE a file */
949                 open_args->opentype = OPEN4_CREATE;
950                 open_args->mode = createmode;
951                 if (createmode == EXCLUSIVE4) {
952                         if (did_excl_setup == FALSE) {
953                                 verf.seconds = zone_get_hostid(NULL);
954                                 if (verf.seconds != 0)
955                                         verf.nseconds = newnum();
956                                 else {
957                                         timestruc_t now;

959                                         gethrestime(&now);
960                                         verf.seconds = now.tv_sec;
961                                         verf.nseconds = now.tv_nsec;
962                                 }
963                                 /*
964                                  * Since the server will use this value for the
965                                  * mtime, make sure that it can't overflow. Zero
966                                  * out the MSB. The actual value does not matter
967                                  * here, only its uniqeness.
968                                  */
969                                 verf.seconds &= INT32_MAX;
970                                 did_excl_setup = TRUE;
971                         }

973                         /* Now copy over verifier to OPEN4args. */
974                         open_args->createhow4_u.createverf = *(uint64_t *)&verf;
975                 } else {
976                         int v_error;
977                         bitmap4 supp_attrs;
978                         servinfo4_t *svp;

980                         attr = &open_args->createhow4_u.createattrs;

982                         svp = drp->r_server;
```



```
983                         (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
984                         supp_attrs = svp->sv_supp_attrs;
985                         nfs_rw_exit(&svp->sv_lock);

987                         /* GUARDED4 or UNCHECKED4 */
988                         v_error = vattr_to_fattr4(in_va, NULL, attr, 0, OP_OPEN,
989                             supp_attrs);
990                         if (v_error) {
991                                 bzero(attr, sizeof (*attr));
992                                 nfs4args_copen_free(open_args);
993                                 nfs4_end_op(VTOMI4(dvp), dvp, vpi,
994                                     &recov_state, FALSE);
995                                 if (ncr != NULL)
996                                         crfree(ncr);
997                                 kmem_free(argop, argoplist_size);
998                                 return (v_error);
999                         }
1000                }
1001        } else {
1002                /* NO CREATE */
1003                open_args->opentype = OPEN4_NOCREATE;
1004        }

1006        if (recov_state.rs_sp != NULL) {
1007                mutex_enter(&recov_state.rs_sp->s_lock);
1008                open_args->owner.clientid = recov_state.rs_sp->clientid;
1009                mutex_exit(&recov_state.rs_sp->s_lock);
1010        } else {
1011                /* XXX should we just fail here? */
1012                open_args->owner.clientid = 0;
1013        }

1015        /*
1016         * This increments oop's ref count or creates a temporary 'just_created'
1017         * open owner that will become valid when this OPEN/OPEN_CONFIRM call
1018         * completes.
1019         */
1020        mutex_enter(&VTOMI4(dvp)->mi_lock);

1022        /* See if a permanent or just created open owner exists */
1023        oop = find_open_owner_nolock(cr, NFS4_JUST_CREATED, VTOMI4(dvp));
1024        if (!oop) {
1025                /*
1026                 * This open owner does not exist so create a temporary
1027                 * just created one.
1028                 */
1029                oop = create_open_owner(cr, VTOMI4(dvp));
1030                ASSERT(oop != NULL);
1031        }
1032        mutex_exit(&VTOMI4(dvp)->mi_lock);

1034        /* this length never changes, do alloc before seqid sync */
1035        open_args->owner.owner_len = sizeof (oop->oo_name);
1036        open_args->owner.owner_val =
1037            kmem_alloc(open_args->owner.owner_len, KM_SLEEP);

1039        e.error = nfs4_start_open_seqid_sync(oop, VTOMI4(dvp));
1040        if (e.error == EAGAIN) {
1041                open_owner_rele(oop);
1042                nfs4args_copen_free(open_args);
1043                nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state, TRUE);
1044                if (ncr != NULL) {
1045                        crfree(ncr);
1046                        ncr = NULL;
1047                }
1048                goto recov_retry;
```

```
1049        }

1051        /* Check to see if we need to do the OTW call */
1052        if (!create_flag) {
1053                if (!nfs4_is_otw_open_necessary(oop, open_flag, vpi,
1054                    file_just_been_created, &e.error, acc, &recov_state)) {

1056                        /*
1057                         * The OTW open is not necessary.  Either
1058                         * the open can succeed without it (eg.
1059                         * delegation, error == 0) or the open
1060                         * must fail due to an access failure
1061                         * (error != 0).  In either case, tidy
1062                         * up and return.
1063                         */

1065                        nfs4_end_open_seqid_sync(oop);
1066                        open_owner_rele(oop);
1067                        nfs4args_copen_free(open_args);
1068                        nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state, FALSE);
1069                        if (ncr != NULL)
1070                                crfree(ncr);
1071                        kmem_free(argop, argoplist_size);
1072                        return (e.error);
1073                }
1074        }

1076        bcopy(&oop->oo_name, open_args->owner.owner_val,
1077            open_args->owner.owner_len);

1079        seqid = nfs4_get_open_seqid(oop) + 1;
1080        open_args->seqid = seqid;
1081        open_args->share_access = 0;
1082        if (open_flag & FREAD)
1083                open_args->share_access |= OPEN4_SHARE_ACCESS_READ;
1084        if (open_flag & FWRITE)
1085                open_args->share_access |= OPEN4_SHARE_ACCESS_WRITE;
1086        open_args->share_deny = OPEN4_SHARE_DENY_NONE;



1090        /*
1091         * getfh w/sanity check for idx_open/idx_fattr
1092         */
1093        ASSERT((idx_open + 1) == (idx_fattr - 1));
1094        argop[idx_open + 1].argop = OP_GETFH;

1096        /* getattr */
1097        argop[idx_fattr].argop = OP_GETATTR;
1098        argop[idx_fattr].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
1099        argop[idx_fattr].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

1101        if (setgid_flag) {
1102                vattr_t _v;
1103                servinfo4_t *svp;
1104                bitmap4 supp_attrs;

1106                svp = drp->r_server;
1107                (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
1108                supp_attrs = svp->sv_supp_attrs;
1109                nfs_rw_exit(&svp->sv_lock);

1111                /*
1112                 * For setgid case, we need to:
1113                 * 4:savefh(new) 5:putfh(dir) 6:getattr(dir) 7:restorefh(new)
1114                 */
```



```
1115                argop[4].argop = OP_SAVEFH;

1117                argop[5].argop = OP_CPUTFH;
1118                argop[5].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

1120                argop[6].argop = OP_GETATTR;
1121                argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
1122                argop[6].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

1124                argop[7].argop = OP_RESTOREFH;

1126                /*
1127                 * nverify
1128                 */
1129                _v.va_mask = AT_GID;
1130                _v.va_gid = in_va->va_gid;
1131                if (!(e.error = nfs4args_verify(&argop[8], &_v, OP_NVERIFY,
1132                    supp_attrs))) {

1134                        /*
1135                         * setattr
1136                         *
1137                         * We _know_ we're not messing with AT_SIZE or
1138                         * AT_XTIME, so no need for stateid or flags.
1139                         * Also we specify NULL rp since we're only
1140                         * interested in setting owner_group attributes.
1141                         */
1142                        nfs4args_setattr(&argop[9], &_v, NULL, 0, NULL, cr,
1143                            supp_attrs, &e.error, 0);
1144                        if (e.error)
1145                                nfs4args_verify_free(&argop[8]);
1146                }

1148                if (e.error) {
1149                        /*
1150                         * XXX - Revisit the last argument to nfs4_end_op()
1151                         *       once 5020486 is fixed.
1152                         */
1153                        nfs4_end_open_seqid_sync(oop);
1154                        open_owner_rele(oop);
1155                        nfs4args_copen_free(open_args);
1156                        nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state, TRUE);
1157                        if (ncr != NULL)
1158                                crfree(ncr);
1159                        kmem_free(argop, argoplist_size);
1160                        return (e.error);
1161                }
1162        } else if (create_flag) {
1163                argop[1].argop = OP_SAVEFH;

1165                argop[5].argop = OP_RESTOREFH;

1167                argop[6].argop = OP_GETATTR;
1168                argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
1169                argop[6].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);
1170        }

1172        NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
1173            "nfs4open_otw: %s call, nm %s, rp %s",
1174            needrecov ? "recov" : "first", file_name,
1175            rnode4info(VTOR4(dvp))));

1177        t = gethrtime();

1179        rfs4call(VTOMI4(dvp), &args, &res, cred_otw, &doqueue, 0, &e);
```

```
1181         if (!e.error && nfs4_need_to_bump_seqid(&res))
1182                 nfs4_set_open_seqid(seqid, oop, args.ctag);
1183
1184         needrecov = nfs4_needs_recovery(&e, TRUE, dvp->v_vfsp);
1185
1186         if (e.error || needrecov) {
1187                 bool_t abort = FALSE;
1188
1189                 if (needrecov) {
1190                         nfs4_bseqid_entry_t *bsep = NULL;
1191
1192                         nfs4open_save_lost_rqst(e.error, &lost_rqst, oop,
1193                             cred_otw, vpi, dvp, open_args);
1194
1195                         if (!e.error && res.status == NFS4ERR_BAD_SEQID) {
1196                                 bsep = nfs4_create_bseqid_entry(oop, NULL,
1197                                     vpi, 0, args.ctag, open_args->seqid);
1198                                 num_bseqid_retry--;
1199                         }
1200
1201                         abort = nfs4_start_recovery(&e, VTOMI4(dvp), dvp, vpi,
1202                             NULL, lost_rqst.lr_op == OP_OPEN ?
1203                             &lost_rqst : NULL, OP_OPEN, bsep, NULL, NULL);
1204
1205                         if (bsep)
1206                                 kmem_free(bsep, sizeof (*bsep));
1207                         /* give up if we keep getting BAD_SEQID */
1208                         if (num_bseqid_retry == 0)
1209                                 abort = TRUE;
1210                         if (abort == TRUE && e.error == 0)
1211                                 e.error = geterrno4(res.status);
1212                 }
1213                 nfs4_end_open_seqid_sync(oop);
1214                 open_owner_rele(oop);
1215                 nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state, needrecov);
1216                 nfs4args_copen_free(open_args);
1217                 if (setgid_flag) {
1218                         nfs4args_verify_free(&argop[8]);
1219                         nfs4args_setattr_free(&argop[9]);
1220                 }
1221                 if (!e.error)
1222                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1223                 if (ncr != NULL) {
1224                         crfree(ncr);
1225                         ncr = NULL;
1226                 }
1227                 if (!needrecov || abort == TRUE || e.error == EINTR ||
1228                     NFS4_FRC_UNMT_ERR(e.error, dvp->v_vfsp)) {
1229                         kmem_free(argop, argoplist_size);
1230                         return (e.error);
1231                 }
1232                 goto recov_retry;
1233         }
1234
1235         /*
1236          * Will check and update lease after checking the rflag for
1237          * OPEN_CONFIRM in the successful OPEN call.
1238          */
1239         if (res.status != NFS4_OK && res.array_len <= idx_fattr + 1) {
1240
1241                 /*
1242                  * XXX what if we're crossing mount points from server1:/drp
1243                  * to server2:/drp/rp.
1244                  */
1245
1246                 /* Signal our end of use of the open seqid */
```



```
1247                 nfs4_end_open_seqid_sync(oop);
1248
1249                 /*
1250                  * This will destroy the open owner if it was just created,
1251                  * and no one else has put a reference on it.
1252                  */
1253                 open_owner_rele(oop);
1254                 if (create_flag && (createmode != EXCLUSIVE4) &&
1255                     res.status == NFS4ERR_BADOWNER)
1256                         nfs4_log_badowner(VTOMI4(dvp), OP_OPEN);
1257
1258                 e.error = geterrno4(res.status);
1259                 nfs4args_copen_free(open_args);
1260                 if (setgid_flag) {
1261                         nfs4args_verify_free(&argop[8]);
1262                         nfs4args_setattr_free(&argop[9]);
1263                 }
1264                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1265                 nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state, needrecov);
1266                 /*
1267                  * If the reply is NFS4ERR_ACCESS, it may be because
1268                  * we are root (no root net access).  If the real uid
1269                  * is not root, then retry with the real uid instead.
1270                  */
1271                 if (ncr != NULL) {
1272                         crfree(ncr);
1273                         ncr = NULL;
1274                 }
1275                 if (res.status == NFS4ERR_ACCESS &&
1276                     (ncr = crnetadjust(cred_otw)) != NULL) {
1277                         cred_otw = ncr;
1278                         goto recov_retry;
1279                 }
1280                 kmem_free(argop, argoplist_size);
1281                 return (e.error);
1282         }
1283
1284         resop = &res.array[idx_open];  /* open res */
1285         op_res = &resop->nfs_resop4_u.opopen;
1286
1287 #ifdef DEBUG
1288         /*
1289          * verify attrset bitmap
1290          */
1291         if (create_flag &&
1292             (createmode == UNCHECKED4 || createmode == GUARDED4)) {
1293                 /* make sure attrset returned is what we asked for */
1294                 /* XXX Ignore this 'error' for now */
1295                 if (attr->attrmask != op_res->attrset)
1296                         /* EMPTY */;
1297         }
1298 #endif
1299
1300         if (op_res->rflags & OPEN4_RESULT_LOCKTYPE_POSIX) {
1301                 mutex_enter(&VTOMI4(dvp)->mi_lock);
1302                 VTOMI4(dvp)->mi_flags |= MI4_POSIX_LOCK;
1303                 mutex_exit(&VTOMI4(dvp)->mi_lock);
1304         }
1305
1306         resop = &res.array[idx_open + 1];  /* getfh res */
1307         gf_res = &resop->nfs_resop4_u.opgetfh;
1308
1309         otw_sfh = sfh4_get(&gf_res->object, VTOMI4(dvp));
1310
1311         /*
1312          * The open stateid has been updated on the server but not
```

```
1313          * on the client yet.  There is a path: makenfs4node->nfs4_attr_cache->
1314          * flush_pages->VOP_PUTPAGE->...->nfs4write where we will issue an OTW
1315          * WRITE call.  That, however, will use the old stateid, so go ahead
1316          * and upate the open stateid now, before any call to makenfs4node.
1317          */
1318         if (vpi) {
1319                 nfs4_open_stream_t      *tmp_osp;
1320                 rnode4_t                *tmp_rp = VTOR4(vpi);

1322                 tmp_osp = find_open_stream(oop, tmp_rp);
1323                 if (tmp_osp) {
1324                         tmp_osp->open_stateid = op_res->stateid;
1325                         mutex_exit(&tmp_osp->os_sync_lock);
1326                         open_stream_rele(tmp_osp, tmp_rp);
1327                 }

1329                 /*
1330                  * We must determine if the file handle given by the otw open
1331                  * is the same as the file handle which was passed in with
1332                  * *vpp.  This case can be reached if the file we are trying
1333                  * to open has been removed and another file has been created
1334                  * having the same file name.  The passed in vnode is released
1335                  * later.
1336                  */
1337                 orig_sfh = VTOR4(vpi)->r_fh;
1338                 fh_differs = nfs4cmpfh(&orig_sfh->sfh_fh, &otw_sfh->sfh_fh);
1339         }

1341         garp = &res.array[idx_fattr].nfs_resop4_u.opgetattr.ga_res;

1343         if (create_flag || fh_differs) {
1344                 int rnode_err = 0;

1346                 vp = makenfs4node(otw_sfh, garp, dvp->v_vfsp, t, cr,
1347                     dvp, fn_get(VTOSV(dvp)->sv_name, file_name, otw_sfh));

1349                 if (e.error)
1350                         PURGE_ATTRCACHE4(vp);
1351                 /*
1352                  * For the newly created vp case, make sure the rnode
1353                  * isn't bad before using it.
1354                  */
1355                 mutex_enter(&(VTOR4(vp))->r_statelock);
1356                 if (VTOR4(vp)->r_flags & R4RECOVERR)
1357                         rnode_err = EIO;
1358                 mutex_exit(&(VTOR4(vp))->r_statelock);

1360                 if (rnode_err) {
1361                         nfs4_end_open_seqid_sync(oop);
1362                         nfs4args_copen_free(open_args);
1363                         if (setgid_flag) {
1364                                 nfs4args_verify_free(&argop[8]);
1365                                 nfs4args_setattr_free(&argop[9]);
1366                         }
1367                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1368                         nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state,
1369                             needrecov);
1370                         open_owner_rele(oop);
1371                         VN_RELE(vp);
1372                         if (ncr != NULL)
1373                                 crfree(ncr);
1374                         sfh4_rele(&otw_sfh);
1375                         kmem_free(argop, argoplist_size);
1376                         return (EIO);
1377                 }
1378         } else {
```



```
1379                 vp = vpi;
1380         }
1381         sfh4_rele(&otw_sfh);

1383         /*
1384          * It seems odd to get a full set of attrs and then not update
1385          * the object's attrcache in the non-create case.  Create case uses
1386          * the attrs since makenfs4node checks to see if the attrs need to
1387          * be updated (and then updates them).  The non-create case should
1388          * update attrs also.
1389          */
1390         if (! create_flag && ! fh_differs && !e.error) {
1391                 nfs4_attr_cache(vp, garp, t, cr, TRUE, NULL);
1392         }

1394         nfs4_error_zinit(&e);
1395         if (op_res->rflags & OPEN4_RESULT_CONFIRM) {
1396                 /* This does not do recovery for vp explicitly. */
1397                 nfs4open_confirm(vp, &seqid, &op_res->stateid, cred_otw, FALSE,
1398                     &retry_open, oop, FALSE, &e, &num_bseqid_retry);

1400                 if (e.error || e.stat) {
1401                         nfs4_end_open_seqid_sync(oop);
1402                         nfs4args_copen_free(open_args);
1403                         if (setgid_flag) {
1404                                 nfs4args_verify_free(&argop[8]);
1405                                 nfs4args_setattr_free(&argop[9]);
1406                         }
1407                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1408                         nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state,
1409                             needrecov);
1410                         open_owner_rele(oop);
1411                         if (create_flag || fh_differs) {
1412                                 /* rele the makenfs4node */
1413                                 VN_RELE(vp);
1414                         }
1415                         if (ncr != NULL) {
1416                                 crfree(ncr);
1417                                 ncr = NULL;
1418                         }
1419                         if (retry_open == TRUE) {
1420                                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
1421                                     "nfs4open_otw: retry the open since OPEN "
1422                                     "CONFIRM failed with error %d stat %d",
1423                                     e.error, e.stat));
1424                                 if (create_flag && createmode == GUARDED4) {
1425                                         NFS4_DEBUG(nfs4_client_recov_debug,
1426                                             (CE_NOTE, "nfs4open_otw: switch "
1427                                             "createmode from GUARDED4 to "
1428                                             "UNCHECKED4"));
1429                                         createmode = UNCHECKED4;
1430                                 }
1431                                 goto recov_retry;
1432                         }
1433                         if (!e.error) {
1434                                 if (create_flag && (createmode != EXCLUSIVE4) &&
1435                                     e.stat == NFS4ERR_BADOWNER)
1436                                         nfs4_log_badowner(VTOMI4(dvp), OP_OPEN);

1438                                 e.error = geterrno4(e.stat);
1439                         }
1440                         kmem_free(argop, argoplist_size);
1441                         return (e.error);
1442                 }
1443         }
```

```
1445         rp = VTOR4(vp);

1447         mutex_enter(&rp->r_statev4_lock);
1448         if (create_flag)
1449                 rp->created_v4 = 1;
1450         mutex_exit(&rp->r_statev4_lock);

1452         mutex_enter(&oop->oo_lock);
1453         /* Doesn't matter if 'oo_just_created' already was set as this */
1454         oop->oo_just_created = NFS4_PERM_CREATED;
1455         if (oop->oo_cred_otw)
1456                 crfree(oop->oo_cred_otw);
1457         oop->oo_cred_otw = cred_otw;
1458         crhold(oop->oo_cred_otw);
1459         mutex_exit(&oop->oo_lock);

1461         /* returns with 'os_sync_lock' held */
1462         osp = find_or_create_open_stream(oop, rp, &created_osp);
1463         if (!osp) {
1464                 NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE,
1465                     "nfs4open_otw: failed to create an open stream"));
1466                 NFS4_DEBUG(nfs4_seqid_sync, (CE_NOTE, "nfs4open_otw: "
1467                     "signal our end of use of the open seqid"));

1469                 nfs4_end_open_seqid_sync(oop);
1470                 open_owner_rele(oop);
1471                 nfs4args_copen_free(open_args);
1472                 if (setgid_flag) {
1473                         nfs4args_verify_free(&argop[8]);
1474                         nfs4args_setattr_free(&argop[9]);
1475                 }
1476                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1477                 nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state, needrecov);
1478                 if (create_flag || fh_differs)
1479                         VN_RELE(vp);
1480                 if (ncr != NULL)
1481                         crfree(ncr);

1483                 kmem_free(argop, argoplist_size);
1484                 return (EINVAL);

1486         }

1488         osp->open_stateid = op_res->stateid;

1490         if (open_flag & FREAD)
1491                 osp->os_share_acc_read++;
1492         if (open_flag & FWRITE)
1493                 osp->os_share_acc_write++;
1494         osp->os_share_deny_none++;

1496         /*
1497          * Need to reset this bitfield for the possible case where we were
1498          * going to OTW CLOSE the file, got a non-recoverable error, and before
1499          * we could retry the CLOSE, OPENed the file again.
1500          */
1501         ASSERT(osp->os_open_owner->oo_seqid_inuse);
1502         osp->os_final_close = 0;
1503         osp->os_force_close = 0;
1504 #ifdef DEBUG
1505         if (osp->os_failed_reopen)
1506                 NFS4_DEBUG(nfs4_open_stream_debug, (CE_NOTE, "nfs4open_otw:"
1507                     " clearing os_failed_reopen for osp %p, cr %p, rp %s",
1508                     (void *)osp, (void *)cr, rnode4info(rp)));
1509 #endif
1510         osp->os_failed_reopen = 0;
```



```
1512         mutex_exit(&osp->os_sync_lock);

1514         nfs4_end_open_seqid_sync(oop);

1516         if (created_osp && recov_state.rs_sp != NULL) {
1517                 mutex_enter(&recov_state.rs_sp->s_lock);
1518                 nfs4_inc_state_ref_count_nolock(recov_state.rs_sp, VTOMI4(dvp));
1519                 mutex_exit(&recov_state.rs_sp->s_lock);
1520         }

1522         /* get rid of our reference to find oop */
1523         open_owner_rele(oop);

1525         open_stream_rele(osp, rp);

1527         /* accept delegation, if any */
1528         nfs4_delegation_accept(rp, CLAIM_NULL, op_res, garp, cred_otw);

1530         nfs4_end_op(VTOMI4(dvp), dvp, vpi, &recov_state, needrecov);

1532         if (createmode == EXCLUSIVE4 &&
1533             (in_va->va_mask & ~(AT_GID | AT_SIZE))) {
1534                 NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE, "nfs4open_otw:"
1535                     " EXCLUSIVE4: sending a SETATTR"));
1536                 /*
1537                  * If doing an exclusive create, then generate
1538                  * a SETATTR to set the initial attributes.
1539                  * Try to set the mtime and the atime to the
1540                  * server's current time.  It is somewhat
1541                  * expected that these fields will be used to
1542                  * store the exclusive create cookie.  If not,
1543                  * server implementors will need to know that
1544                  * a SETATTR will follow an exclusive create
1545                  * and the cookie should be destroyed if
1546                  * appropriate.
1547                  *
1548                  * The AT_GID and AT_SIZE bits are turned off
1549                  * so that the SETATTR request will not attempt
1550                  * to process these.  The gid will be set
1551                  * separately if appropriate.  The size is turned
1552                  * off because it is assumed that a new file will
1553                  * be created empty and if the file wasn't empty,
1554                  * then the exclusive create will have failed
1555                  * because the file must have existed already.
1556                  * Therefore, no truncate operation is needed.
1557                  */
1558                 in_va->va_mask &= ~(AT_GID | AT_SIZE);
1559                 in_va->va_mask |= (AT_MTIME | AT_ATIME);

1561                 e.error = nfs4setattr(vp, in_va, 0, cr, NULL);
1562                 if (e.error) {
1563                         /*
1564                          * Couldn't correct the attributes of
1565                          * the newly created file and the
1566                          * attributes are wrong.  Remove the
1567                          * file and return an error to the
1568                          * application.
1569                          */
1570                         /* XXX will this take care of client state ? */
1571                         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE,
1572                             "nfs4open_otw: EXCLUSIVE4: error %d on SETATTR:"
1573                             " remove file", e.error));
1574                         VN_RELE(vp);
1575                         (void) nfs4_remove(dvp, file_name, cr, NULL, 0);
1576                         /*
```

```
1577                         * Since we've reled the vnode and removed
1578                         * the file we now need to return the error.
1579                         * At this point we don't want to update the
1580                         * dircaches, call nfs4_waitfor_purge_complete
1581                         * or set vpp to vp so we need to skip these
1582                         * as well.
1583                         */
1584                        goto skip_update_dircaches;
1585                }
1586        }
1587
1588        /*
1589         * If we created or found the correct vnode, due to create_flag or
1590         * fh_differs being set, then update directory cache attribute, readdir
1591         * and dnlc caches.
1592         */
1593        if (create_flag || fh_differs) {
1594                dirattr_info_t dinfo, *dinfop;
1595
1596                /*
1597                 * Make sure getattr succeeded before using results.
1598                 * note: op 7 is getattr(dir) for both flavors of
1599                 * open(create).
1600                 */
1601                if (create_flag && res.status == NFS4_OK) {
1602                        dinfo.di_time_call = t;
1603                        dinfo.di_cred = cr;
1604                        dinfo.di_garp =
1605                            &res.array[6].nfs_resop4_u.opgetattr.ga_res;
1606                        dinfop = &dinfo;
1607                } else {
1608                        dinfop = NULL;
1609                }
1610
1611                nfs4_update_dircaches(&op_res->cinfo, dvp, vp, file_name,
1612                    dinfop);
1613        }
1614
1615        /*
1616         * If the page cache for this file was flushed from actions
1617         * above, it was done asynchronously and if that is true,
1618         * there is a need to wait here for it to complete.  This must
1619         * be done outside of start_fop/end_fop.
1620         */
1621        (void) nfs4_waitfor_purge_complete(vp);
1622
1623        /*
1624         * It is implicit that we are in the open case (create_flag == 0) since
1625         * fh_differs can only be set to a non-zero value in the open case.
1626         */
1627        if (fh_differs != 0 && vpi != NULL)
1628                VN_RELE(vpi);
1629
1630        /*
1631         * Be sure to set *vpp to the correct value before returning.
1632         */
1633        *vpp = vp;
1634
1635 skip_update_dircaches:
1636
1637        nfs4args_copen_free(open_args);
1638        if (setgid_flag) {
1639                nfs4args_verify_free(&argop[8]);
1640                nfs4args_setattr_free(&argop[9]);
1641        }
1642        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
```



```
1644        if (ncr)
1645                crfree(ncr);
1646        kmem_free(argop, argoplist_size);
1647        return (e.error);
1648 }
1649
1650 /*
1651  * Reopen an open instance.  cf. nfs4open_otw().
1652  *
1653  * Errors are returned by the nfs4_error_t parameter.
1654  * - ep->error contains an errno value or zero.
1655  * - if it is zero, ep->stat is set to an NFS status code, if any.
1656  *   If the file could not be reopened, but the caller should continue, the
1657  *   file is marked dead and no error values are returned.  If the caller
1658  *   should stop recovering open files and start over, either the ep->error
1659  *   value or ep->stat will indicate an error (either something that requires
1660  *   recovery or EAGAIN).  Note that some recovery (e.g., expired volatile
1661  *   filehandles) may be handled silently by this routine.
1662  * - if it is EINTR, ETIMEDOUT, or NFS4_FRC_UNMT_ERR, recovery for lost state
1663  *   will be started, so the caller should not do it.
1664  *
1665  * Gotos:
1666  * - kill_file : reopen failed in such a fashion to constitute marking the
1667  *    file dead and setting the open stream's 'os_failed_reopen' as 1.  This
1668  *   is for cases where recovery is not possible.
1669  * - failed_reopen : same as above, except that the file has already been
1670  *   marked dead, so no need to do it again.
1671  * - bailout : reopen failed but we are able to recover and retry the reopen -
1672  *   either within this function immediately or via the calling function.
1673  */
1674
1675 void
1676 nfs4_reopen(vnode_t *vp, nfs4_open_stream_t *osp, nfs4_error_t *ep,
1677     open_claim_type4 claim, bool_t frc_use_claim_previous,
1678     bool_t is_recov)
1679 {
1680         COMPOUND4args_clnt args;
1681         COMPOUND4res_clnt res;
1682         nfs_argop4 argop[4];
1683         nfs_resop4 *resop;
1684         OPEN4res *op_res = NULL;
1685         OPEN4cargs *open_args;
1686         GETFH4res *gf_res;
1687         rnode4_t *rp = VTOR4(vp);
1688         int doqueue = 1;
1689         cred_t *cr = NULL, *cred_otw = NULL;
1690         nfs4_open_owner_t *oop = NULL;
1691         seqid4 seqid;
1692         nfs4_ga_res_t *garp;
1693         char fn[MAXNAMELEN];
1694         nfs4_recov_state_t recov = {NULL, 0};
1695         nfs4_lost_rqst_t lost_rqst;
1696         mntinfo4_t *mi = VTOMI4(vp);
1697         bool_t abort;
1698         char *failed_msg = "";
1699         int fh_different;
1700         hrtime_t t;
1701         nfs4_bseqid_entry_t *bsep = NULL;
1702
1703         ASSERT(nfs4_consistent_type(vp));
1704         ASSERT(nfs_zone() == mi->mi_zone);
1705
1706         nfs4_error_zinit(ep);
1707
1708         /* this is the cred used to find the open owner */
```

```
1709         cr = state_to_cred(osp);
1710         if (cr == NULL) {
1711                 failed_msg = "Couldn't reopen: no cred";
1712                 goto kill_file;
1713         }
1714         /* use this cred for OTW operations */
1715         cred_otw = nfs4_get_otw_cred(cr, mi, osp->os_open_owner);

1717 top:
1718         nfs4_error_zinit(ep);

1720         if (mi->mi_vfsp->vfs_flag & VFS_UNMOUNTED) {
1721                 /* File system has been unmounted, quit */
1722                 ep->error = EIO;
1723                 failed_msg = "Couldn't reopen: file system has been unmounted";
1724                 goto kill_file;
1725         }

1727         oop = osp->os_open_owner;

1729         ASSERT(oop != NULL);
1730         if (oop == NULL) {      /* be defensive in non-DEBUG */
1731                 failed_msg = "can't reopen: no open owner";
1732                 goto kill_file;
1733         }
1734         open_owner_hold(oop);

1736         ep->error = nfs4_start_open_seqid_sync(oop, mi);
1737         if (ep->error) {
1738                 open_owner_rele(oop);
1739                 oop = NULL;
1740                 goto bailout;
1741         }

1743         /*
1744          * If the rnode has a delegation and the delegation has been
1745          * recovered and the server didn't request a recall and the caller
1746          * didn't specifically ask for CLAIM_PREVIOUS (nfs4frlock during
1747          * recovery) and the rnode hasn't been marked dead, then install
1748          * the delegation stateid in the open stream.  Otherwise, proceed
1749          * with a CLAIM_PREVIOUS or CLAIM_NULL OPEN.
1750          */
1751         mutex_enter(&rp->r_statev4_lock);
1752         if (rp->r_deleg_type != OPEN_DELEGATE_NONE &&
1753             !rp->r_deleg_return_pending &&
1754             (rp->r_deleg_needs_recovery == OPEN_DELEGATE_NONE) &&
1755             !rp->r_deleg_needs_recall &&
1756             claim != CLAIM_DELEGATE_CUR && !frc_use_claim_previous &&
1757             !(rp->r_flags & R4RECOVERR)) {
1758                 mutex_enter(&osp->os_sync_lock);
1759                 osp->os_delegation = 1;
1760                 osp->open_stateid = rp->r_deleg_stateid;
1761                 mutex_exit(&osp->os_sync_lock);
1762                 mutex_exit(&rp->r_statev4_lock);
1763                 goto bailout;
1764         }
1765         mutex_exit(&rp->r_statev4_lock);

1767         /*
1768          * If the file failed recovery, just quit.  This failure need not
1769          * affect other reopens, so don't return an error.
1770          */
1771         mutex_enter(&rp->r_statelock);
1772         if (rp->r_flags & R4RECOVERR) {
1773                 mutex_exit(&rp->r_statelock);
1774                 ep->error = 0;
```



```
1775                 goto failed_reopen;
1776         }
1777         mutex_exit(&rp->r_statelock);

1779         /*
1780          * argop is empty here
1781          *
1782          * PUTFH, OPEN, GETATTR
1783          */
1784         args.ctag = TAG_REOPEN;
1785         args.array_len = 4;
1786         args.array = argop;

1788         NFS4_DEBUG(nfs4_client_failover_debug, (CE_NOTE,
1789             "nfs4_reopen: file is type %d, id %s",
1790             vp->v_type, rnode4info(VTOR4(vp))));

1792         argop[0].argop = OP_CPUTFH;

1794         if (claim != CLAIM_PREVIOUS) {
1795                 /*
1796                  * if this is a file mount then
1797                  * use the mntinfo parentfh
1798                  */
1799                 argop[0].nfs_argop4_u.opcputfh.sfh =
1800                     (vp->v_flag & VROOT) ? mi->mi_srvparentfh :
1801                     VTOSV(vp)->sv_dfh;
1802         } else {
1803                 /* putfh fh to reopen */
1804                 argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;
1805         }

1807         argop[1].argop = OP_COPEN;
1808         open_args = &argop[1].nfs_argop4_u.opcopen;
1809         open_args->claim = claim;

1811         if (claim == CLAIM_NULL) {

1813                 if ((ep->error = vtoname(vp, fn, MAXNAMELEN)) != 0) {
1814                         nfs_cmn_err(ep->error, CE_WARN, "nfs4_reopen: vtoname "
1815                             "failed for vp 0x%p for CLAIM_NULL with %m",
1816                             (void *)vp);
1817                         failed_msg = "Couldn't reopen: vtoname failed for "
1818                             "CLAIM_NULL";
1819                         /* nothing allocated yet */
1820                         goto kill_file;
1821                 }

1823                 open_args->open_claim4_u.cfile = fn;
1824         } else if (claim == CLAIM_PREVIOUS) {

1826                 /*
1827                  * We have two cases to deal with here:
1828                  * 1) We're being called to reopen files in order to satisfy
1829                  *    a lock operation request which requires us to explicitly
1830                  *    reopen files which were opened under a delegation.  If
1831                  *    we're in recovery, we *must* use CLAIM_PREVIOUS.  In
1832                  *    that case, frc_use_claim_previous is TRUE and we must
1833                  *    use the rnode's current delegation type (r_deleg_type).
1834                  * 2) We're reopening files during some form of recovery.
1835                  *    In this case, frc_use_claim_previous is FALSE and we
1836                  *    use the delegation type appropriate for recovery
1837                  *    (r_deleg_needs_recovery).
1838                  */
1839                 mutex_enter(&rp->r_statev4_lock);
1840                 open_args->open_claim4_u.delegate_type =
```

```
1841                    frc_use_claim_previous ?
1842                    rp->r_deleg_type :
1843                    rp->r_deleg_needs_recovery;
1844                mutex_exit(&rp->r_statev4_lock);

1846        } else if (claim == CLAIM_DELEGATE_CUR) {

1848                if ((ep->error = vtoname(vp, fn, MAXNAMELEN)) != 0) {
1849                        nfs_cmn_err(ep->error, CE_WARN, "nfs4_reopen: vtoname "
1850                            "failed for vp 0x%p for CLAIM_DELEGATE_CUR "
1851                            "with %m", (void *)vp);
1852                        failed_msg = "Couldn't reopen: vtoname failed for "
1853                            "CLAIM_DELEGATE_CUR";
1854                        /* nothing allocated yet */
1855                        goto kill_file;
1856                }

1858                mutex_enter(&rp->r_statev4_lock);
1859                open_args->open_claim4_u.delegate_cur_info.delegate_stateid =
1860                    rp->r_deleg_stateid;
1861                mutex_exit(&rp->r_statev4_lock);

1863                open_args->open_claim4_u.delegate_cur_info.cfile = fn;
1864        }
1865        open_args->opentype = OPEN4_NOCREATE;
1866        open_args->owner.clientid = mi2clientid(mi);
1867        open_args->owner.owner_len = sizeof (oop->oo_name);
1868        open_args->owner.owner_val =
1869            kmem_alloc(open_args->owner.owner_len, KM_SLEEP);
1870        bcopy(&oop->oo_name, open_args->owner.owner_val,
1871            open_args->owner.owner_len);
1872        open_args->share_access = 0;
1873        open_args->share_deny = 0;

1875        mutex_enter(&osp->os_sync_lock);
1876        NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE, "nfs4_reopen: osp %p rp "
1877            "%p: read acc %"PRIu64" write acc %"PRIu64": open ref count %d: "
1878            "mmap read %"PRIu64" mmap write %"PRIu64" claim %d ",
1879            (void *)osp, (void *)rp, osp->os_share_acc_read,
1880            osp->os_share_acc_write, osp->os_open_ref_count,
1881            osp->os_mmap_read, osp->os_mmap_write, claim));

1883        if (osp->os_share_acc_read || osp->os_mmap_read)
1884                open_args->share_access |= OPEN4_SHARE_ACCESS_READ;
1885        if (osp->os_share_acc_write || osp->os_mmap_write)
1886                open_args->share_access |= OPEN4_SHARE_ACCESS_WRITE;
1887        if (osp->os_share_deny_read)
1888                open_args->share_deny |= OPEN4_SHARE_DENY_READ;
1889        if (osp->os_share_deny_write)
1890                open_args->share_deny |= OPEN4_SHARE_DENY_WRITE;
1891        mutex_exit(&osp->os_sync_lock);

1893        seqid = nfs4_get_open_seqid(oop) + 1;
1894        open_args->seqid = seqid;

1896        /* Construct the getfh part of the compound */
1897        argop[2].argop = OP_GETFH;

1899        /* Construct the getattr part of the compound */
1900        argop[3].argop = OP_GETATTR;
1901        argop[3].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
1902        argop[3].nfs_argop4_u.opgetattr.mi = mi;

1904        t = gethrtime();

1906        rfs4call(mi, &args, &res, cred_otw, &doqueue, 0, ep);
```



```
1908        if (ep->error) {
1909                if (!is_recov && !frc_use_claim_previous &&
1910                    (ep->error == EINTR || ep->error == ETIMEDOUT ||
1911                    NFS4_FRC_UNMT_ERR(ep->error, vp->v_vfsp))) {
1912                        nfs4open_save_lost_rqst(ep->error, &lost_rqst, oop,
1913                            cred_otw, vp, NULL, open_args);
1914                        abort = nfs4_start_recovery(ep,
1915                            VTOMI4(vp), vp, NULL, NULL,
1916                            lost_rqst.lr_op == OP_OPEN ?
1917                            &lost_rqst : NULL, OP_OPEN, NULL, NULL, NULL);
1918                        nfs4args_copen_free(open_args);
1919                        goto bailout;
1920                }

1922                nfs4args_copen_free(open_args);

1924                if (ep->error == EACCES && cred_otw != cr) {
1925                        crfree(cred_otw);
1926                        cred_otw = cr;
1927                        crhold(cred_otw);
1928                        nfs4_end_open_seqid_sync(oop);
1929                        open_owner_rele(oop);
1930                        oop = NULL;
1931                        goto top;
1932                }
1933                if (ep->error == ETIMEDOUT)
1934                        goto bailout;
1935                failed_msg = "Couldn't reopen: rpc error";
1936                goto kill_file;
1937        }

1939        if (nfs4_need_to_bump_seqid(&res))
1940                nfs4_set_open_seqid(seqid, oop, args.ctag);

1942        switch (res.status) {
1943        case NFS4_OK:
1944                if (recov.rs_flags & NFS4_RS_DELAY_MSG) {
1945                        mutex_enter(&rp->r_statelock);
1946                        rp->r_delay_interval = 0;
1947                        mutex_exit(&rp->r_statelock);
1948                }
1949                break;
1950        case NFS4ERR_BAD_SEQID:
1951                bsep = nfs4_create_bseqid_entry(oop, NULL, vp, 0,
1952                    args.ctag, open_args->seqid);

1954                abort = nfs4_start_recovery(ep, VTOMI4(vp), vp, NULL,
1955                    NULL, lost_rqst.lr_op == OP_OPEN ? &lost_rqst :
1956                    NULL, OP_OPEN, bsep, NULL, NULL);

1958                nfs4args_copen_free(open_args);
1959                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1960                nfs4_end_open_seqid_sync(oop);
1961                open_owner_rele(oop);
1962                oop = NULL;
1963                kmem_free(bsep, sizeof (*bsep));

1965                goto kill_file;
1966        case NFS4ERR_NO_GRACE:
1967                nfs4args_copen_free(open_args);
1968                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1969                nfs4_end_open_seqid_sync(oop);
1970                open_owner_rele(oop);
1971                oop = NULL;
1972                if (claim == CLAIM_PREVIOUS) {
```

```
1973                        /*
1974                         * Retry as a plain open. We don't need to worry about
1975                         * checking the changeinfo: it is acceptable for a
1976                         * client to re-open a file and continue processing
1977                         * (in the absence of locks).
1978                         */
1979                        NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
1980                            "nfs4_reopen: CLAIM_PREVIOUS: NFS4ERR_NO_GRACE; "
1981                            "will retry as CLAIM_NULL"));
1982                        claim = CLAIM_NULL;
1983                        nfs4_mi_kstat_inc_no_grace(mi);
1984                        goto top;
1985                }
1986                failed_msg =
1987                    "Couldn't reopen: tried reclaim outside grace period. ";
1988                goto kill_file;
1989        case NFS4ERR_GRACE:
1990                nfs4_set_grace_wait(mi);
1991                nfs4args_copen_free(open_args);
1992                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
1993                nfs4_end_open_seqid_sync(oop);
1994                open_owner_rele(oop);
1995                oop = NULL;
1996                ep->error = nfs4_wait_for_grace(mi, &recov);
1997                if (ep->error != 0)
1998                        goto bailout;
1999                goto top;
2000        case NFS4ERR_DELAY:
2001                nfs4_set_delay_wait(vp);
2002                nfs4args_copen_free(open_args);
2003                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2004                nfs4_end_open_seqid_sync(oop);
2005                open_owner_rele(oop);
2006                oop = NULL;
2007                ep->error = nfs4_wait_for_delay(vp, &recov);
2008                nfs4_mi_kstat_inc_delay(mi);
2009                if (ep->error != 0)
2010                        goto bailout;
2011                goto top;
2012        case NFS4ERR_FHEXPIRED:
2013                /* recover filehandle and retry */
2014                abort = nfs4_start_recovery(ep,
2015                    mi, vp, NULL, NULL, NULL, OP_OPEN, NULL, NULL, NULL);
2016                nfs4args_copen_free(open_args);
2017                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2018                nfs4_end_open_seqid_sync(oop);
2019                open_owner_rele(oop);
2020                oop = NULL;
2021                if (abort == FALSE)
2022                        goto top;
2023                failed_msg = "Couldn't reopen: recovery aborted";
2024                goto kill_file;
2025        case NFS4ERR_RESOURCE:
2026        case NFS4ERR_STALE_CLIENTID:
2027        case NFS4ERR_WRONGSEC:
2028        case NFS4ERR_EXPIRED:
2029                /*
2030                 * Do not mark the file dead and let the calling
2031                 * function initiate recovery.
2032                 */
2033                nfs4args_copen_free(open_args);
2034                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2035                nfs4_end_open_seqid_sync(oop);
2036                open_owner_rele(oop);
2037                oop = NULL;
2038                goto bailout;
```



```
2039        case NFS4ERR_ACCESS:
2040                if (cred_otw != cr) {
2041                        crfree(cred_otw);
2042                        cred_otw = cr;
2043                        crhold(cred_otw);
2044                        nfs4args_copen_free(open_args);
2045                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2046                        nfs4_end_open_seqid_sync(oop);
2047                        open_owner_rele(oop);
2048                        oop = NULL;
2049                        goto top;
2050                }
2051                /* fall through */
2052        default:
2053                NFS4_DEBUG(nfs4_client_failover_debug, (CE_NOTE,
2054                    "nfs4_reopen: r_server 0x%p, mi_curr_serv 0x%p, rnode %s",
2055                    (void*)VTOR4(vp)->r_server, (void*)mi->mi_curr_serv,
2056                    rnode4info(VTOR4(vp))));
2057                failed_msg = "Couldn't reopen: NFSv4 error";
2058                nfs4args_copen_free(open_args);
2059                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2060                goto kill_file;
2061        }

2063        resop = &res.array[1];  /* open res */
2064        op_res = &resop->nfs_resop4_u.opopen;

2066        garp = &res.array[3].nfs_resop4_u.opgetattr.ga_res;

2068        /*
2069         * Check if the path we reopened really is the same
2070         * file. We could end up in a situation where the file
2071         * was removed and a new file created with the same name.
2072         */
2073        resop = &res.array[2];
2074        gf_res = &resop->nfs_resop4_u.opgetfh;
2075        (void) nfs_rw_enter_sig(&mi->mi_fh_lock, RW_READER, 0);
2076        fh_different = (nfs4cmpfh(&rp->r_fh->sfh_fh, &gf_res->object) != 0);
2077        if (fh_different) {
2078                if (mi->mi_fh_expire_type == FH4_PERSISTENT ||
2079                    mi->mi_fh_expire_type & FH4_NOEXPIRE_WITH_OPEN) {
2080                        /* Oops, we don't have the same file */
2081                        if (mi->mi_fh_expire_type == FH4_PERSISTENT)
2082                                failed_msg = "Couldn't reopen: Persistent "
2083                                    "file handle changed";
2084                        else
2085                                failed_msg = "Couldn't reopen: Volatile "
2086                                    "(no expire on open) file handle changed";

2088                        nfs4args_copen_free(open_args);
2089                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2090                        nfs_rw_exit(&mi->mi_fh_lock);
2091                        goto kill_file;

2093                } else {
2094                        /*
2095                         * We have volatile file handles that don't compare.
2096                         * If the fids are the same then we assume that the
2097                         * file handle expired but the rnode still refers to
2098                         * the same file object.
2099                         *
2100                         * First check that we have fids or not.
2101                         * If we don't we have a dumb server so we will
2102                         * just assume every thing is ok for now.
2103                         */
2104                        if (!ep->error && garp->n4g_va.va_mask & AT_NODEID &&
```

```
2105                            rp->r_attr.va_mask & AT_NODEID &&
2106                            rp->r_attr.va_nodeid != garp->n4g_va.va_nodeid) {
2107                                /*
2108                                 * We have fids, but they don't
2109                                 * compare. So kill the file.
2110                                 */
2111                                failed_msg =
2112                                    "Couldn't reopen: file handle changed"
2113                                    " due to mismatched fids";
2114                                nfs4args_copen_free(open_args);
2115                                (void) xdr_free(xdr_COMPOUND4res_clnt,
2116                                    (caddr_t)&res);
2117                                nfs_rw_exit(&mi->mi_fh_lock);
2118                                goto kill_file;
2119                        } else {
2120                                /*
2121                                 * We have volatile file handles that refers
2122                                 * to the same file (at least they have the
2123                                 * same fid) or we don't have fids so we
2124                                 * can't tell. :(. We'll be a kind and accepting
2125                                 * client so we'll update the rnode's file
2126                                 * handle with the otw handle.
2127                                 *
2128                                 * We need to drop mi->mi_fh_lock since
2129                                 * sh4_update acquires it. Since there is
2130                                 * only one recovery thread there is no
2131                                 * race.
2132                                 */
2133                                nfs_rw_exit(&mi->mi_fh_lock);
2134                                sfh4_update(rp->r_fh, &gf_res->object);
2135                        }
2136                }
2137        } else {
2138                nfs_rw_exit(&mi->mi_fh_lock);
2139        }

2141        ASSERT(nfs4_consistent_type(vp));

2143        /*
2144         * If the server wanted an OPEN_CONFIRM but that fails, just start
2145         * over.  Presumably if there is a persistent error it will show up
2146         * when we resend the OPEN.
2147         */
2148        if (op_res->rflags & OPEN4_RESULT_CONFIRM) {
2149                bool_t retry_open = FALSE;

2151                nfs4open_confirm(vp, &seqid, &op_res->stateid,
2152                    cred_otw, is_recov, &retry_open,
2153                    oop, FALSE, ep, NULL);
2154                if (ep->error || ep->stat) {
2155                        nfs4args_copen_free(open_args);
2156                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2157                        nfs4_end_open_seqid_sync(oop);
2158                        open_owner_rele(oop);
2159                        oop = NULL;
2160                        goto top;
2161                }
2162        }

2164        mutex_enter(&osp->os_sync_lock);
2165        osp->open_stateid = op_res->stateid;
2166        osp->os_delegation = 0;
2167        /*
2168         * Need to reset this bitfield for the possible case where we were
2169         * going to OTW CLOSE the file, got a non-recoverable error, and before
2170         * we could retry the CLOSE, OPENed the file again.
```



```
2171         */
2172        ASSERT(osp->os_open_owner->oo_seqid_inuse);
2173        osp->os_final_close = 0;
2174        osp->os_force_close = 0;
2175        if (claim == CLAIM_DELEGATE_CUR || claim == CLAIM_PREVIOUS)
2176                osp->os_dc_openacc = open_args->share_access;
2177        mutex_exit(&osp->os_sync_lock);

2179        nfs4_end_open_seqid_sync(oop);

2181        /* accept delegation, if any */
2182        nfs4_delegation_accept(rp, claim, op_res, garp, cred_otw);

2184        nfs4args_copen_free(open_args);

2186        nfs4_attr_cache(vp, garp, t, cr, TRUE, NULL);

2188        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

2190        ASSERT(nfs4_consistent_type(vp));

2192        open_owner_rele(oop);
2193        crfree(cr);
2194        crfree(cred_otw);
2195        return;

2197 kill_file:
2198        nfs4_fail_recov(vp, failed_msg, ep->error, ep->stat);
2199 failed_reopen:
2200        NFS4_DEBUG(nfs4_open_stream_debug, (CE_NOTE,
2201            "nfs4_reopen: setting os_failed_reopen for osp %p, cr %p, rp %s",
2202            (void *)osp, (void *)cr, rnode4info(rp)));
2203        mutex_enter(&osp->os_sync_lock);
2204        osp->os_failed_reopen = 1;
2205        mutex_exit(&osp->os_sync_lock);
2206 bailout:
2207        if (oop != NULL) {
2208                nfs4_end_open_seqid_sync(oop);
2209                open_owner_rele(oop);
2210        }
2211        if (cr != NULL)
2212                crfree(cr);
2213        if (cred_otw != NULL)
2214                crfree(cred_otw);
2215 }

2217 /* for . and .. OPENs */
2218 /* ARGSUSED */
2219 static int
2220 nfs4_open_non_reg_file(vnode_t **vpp, int flag, cred_t *cr)
2221 {
2222        rnode4_t *rp;
2223        nfs4_ga_res_t gar;

2225        ASSERT(nfs_zone() == VTOMI4(*vpp)->mi_zone);

2227        /*
2228         * If close-to-open consistency checking is turned off or
2229         * if there is no cached data, we can avoid
2230         * the over the wire getattr.  Otherwise, force a
2231         * call to the server to get fresh attributes and to
2232         * check caches. This is required for close-to-open
2233         * consistency.
2234         */
2235        rp = VTOR4(*vpp);
2236        if (VTOMI4(*vpp)->mi_flags & MI4_NOCTO ||
```

```
2237             (rp->r_dir == NULL && !nfs4_has_pages(*vpp)))
2238                 return (0);
2239
2240         gar.n4g_va.va_mask = AT_ALL;
2241         return (nfs4_getattr_otw(*vpp, &gar, cr, 0));
2242 }
2243
2244 /*
2245  * CLOSE a file
2246  */
2247 /* ARGSUSED */
2248 static int
2249 nfs4_close(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
2250         caller_context_t *ct)
2251 {
2252         rnode4_t        *rp;
2253         int              error = 0;
2254         int              r_error = 0;
2255         int              n4error = 0;
2256         nfs4_error_t     e = { 0, NFS4_OK, RPC_SUCCESS };
2257
2258         /*
2259          * Remove client state for this (lockowner, file) pair.
2260          * Issue otw v4 call to have the server do the same.
2261          */
2262
2263         rp = VTOR4(vp);
2264
2265         /*
2266          * zone_enter(2) prevents processes from changing zones with NFS files
2267          * open; if we happen to get here from the wrong zone we can't do
2268          * anything over the wire.
2269          */
2270         if (VTOMI4(vp)->mi_zone != nfs_zone()) {
2271                 /*
2272                  * We could attempt to clean up locks, except we're sure
2273                  * that the current process didn't acquire any locks on
2274                  * the file: any attempt to lock a file belong to another zone
2275                  * will fail, and one can't lock an NFS file and then change
2276                  * zones, as that fails too.
2277                  *
2278                  * Returning an error here is the sane thing to do.  A
2279                  * subsequent call to VN_RELE() which translates to a
2280                  * nfs4_inactive() will clean up state: if the zone of the
2281                  * vnode's origin is still alive and kicking, the inactive
2282                  * thread will handle the request (from the correct zone), and
2283                  * everything (minus the OTW close call) should be OK.  If the
2284                  * zone is going away nfs4_async_inactive() will throw away
2285                  * delegations, open streams and cached pages inline.
2286                  */
2287                 return (EIO);
2288         }
2289
2290         /*
2291          * If we are using local locking for this filesystem, then
2292          * release all of the SYSV style record locks.  Otherwise,
2293          * we are doing network locking and we need to release all
2294          * of the network locks.  All of the locks held by this
2295          * process on this file are released no matter what the
2296          * incoming reference count is.
2297          */
2298         if (VTOMI4(vp)->mi_flags & MI4_LLOCK) {
2299                 cleanlocks(vp, ttoproc(curthread)->p_pid, 0);
2300                 cleanshares(vp, ttoproc(curthread)->p_pid);
2301         } else
2302                 e.error = nfs4_lockrelease(vp, flag, offset, cr);
```



```
2303
2304         if (e.error) {
2305                 struct lm_sysid *lmsid;
2306                 lmsid = nfs4_find_sysid(VTOMI4(vp));
2307                 if (lmsid == NULL) {
2308                         DTRACE_PROBE2(unknown__sysid, int, e.error,
2309                             vnode_t *, vp);
2310                 } else {
2311                         cleanlocks(vp, ttoproc(curthread)->p_pid,
2312                             (lm_sysidt(lmsid) | LM_SYSID_CLIENT));
2313                 }
2314                 return (e.error);
2315         }
2316
2317         if (count > 1)
2318                 return (0);
2319
2320         /*
2321          * If the file has been 'unlinked', then purge the
2322          * DNLC so that this vnode will get reycled quicker
2323          * and the .nfs* file on the server will get removed.
2324          */
2325         if (rp->r_unldvp != NULL)
2326                 dnlc_purge_vp(vp);
2327
2328         /*
2329          * If the file was open for write and there are pages,
2330          * do a synchronous flush and commit of all of the
2331          * dirty and uncommitted pages.
2332          */
2333         ASSERT(!e.error);
2334         if ((flag & FWRITE) && nfs4_has_pages(vp))
2335                 error = nfs4_putpage_commit(vp, 0, 0, cr);
2336
2337         mutex_enter(&rp->r_statelock);
2338         r_error = rp->r_error;
2339         rp->r_error = 0;
2340         mutex_exit(&rp->r_statelock);
2341
2342         /*
2343          * If this file type is one for which no explicit 'open' was
2344          * done, then bail now (ie. no need for protocol 'close'). If
2345          * there was an error w/the vm subsystem, return _that_ error,
2346          * otherwise, return any errors that may've been reported via
2347          * the rnode.
2348          */
2349         if (vp->v_type != VREG)
2350                 return (error ? error : r_error);
2351
2352         /*
2353          * The sync putpage commit may have failed above, but since
2354          * we're working w/a regular file, we need to do the protocol
2355          * 'close' (nfs4close_one will figure out if an otw close is
2356          * needed or not). Report any errors _after_ doing the protocol
2357          * 'close'.
2358          */
2359         nfs4close_one(vp, NULL, cr, flag, NULL, &e, CLOSE_NORM, 0, 0, 0);
2360         n4error = e.error ? e.error : geterrno4(e.stat);
2361
2362         /*
2363          * Error reporting prio (Hi -> Lo)
2364          *
2365          *   i) nfs4_putpage_commit (error)
2366          *  ii) rnode's (r_error)
2367          * iii) nfs4close_one (n4error)
2368          */
```

```
2369         return (error ? error : (r_error ? r_error : n4error));
2370 }

2372 /*
2373  * Initialize *lost_rqstp.
2374  */

2376 static void
2377 nfs4close_save_lost_rqst(int error, nfs4_lost_rqst_t *lost_rqstp,
2378     nfs4_open_owner_t *oop, nfs4_open_stream_t *osp, cred_t *cr,
2379     vnode_t *vp)
2380 {
2381         if (error != ETIMEDOUT && error != EINTR &&
2382             !NFS4_FRC_UNMT_ERR(error, vp->v_vfsp)) {
2383                 lost_rqstp->lr_op = 0;
2384                 return;
2385         }

2387         NFS4_DEBUG(nfs4_lost_rqst_debug, (CE_NOTE,
2388             "nfs4close_save_lost_rqst: error %d", error));

2390         lost_rqstp->lr_op = OP_CLOSE;
2391         /*
2392          * The vp is held and rele'd via the recovery code.
2393          * See nfs4_save_lost_rqst.
2394          */
2395         lost_rqstp->lr_vp = vp;
2396         lost_rqstp->lr_dvp = NULL;
2397         lost_rqstp->lr_oop = oop;
2398         lost_rqstp->lr_osp = osp;
2399         ASSERT(osp != NULL);
2400         ASSERT(mutex_owned(&osp->os_sync_lock));
2401         osp->os_pending_close = 1;
2402         lost_rqstp->lr_lop = NULL;
2403         lost_rqstp->lr_cr = cr;
2404         lost_rqstp->lr_flk = NULL;
2405         lost_rqstp->lr_putfirst = FALSE;
2406 }

2408 /*
2409  * Assumes you already have the open seqid sync grabbed as well as the
2410  * 'os_sync_lock'.  Note: this will release the open seqid sync and
2411  * 'os_sync_lock' if client recovery starts.  Calling functions have to
2412  * be prepared to handle this.
2413  *
2414  * 'recov' is returned as 1 if the CLOSE operation detected client recovery
2415  * was needed and was started, and that the calling function should retry
2416  * this function; otherwise it is returned as 0.
2417  *
2418  * Errors are returned via the nfs4_error_t parameter.
2419  */
2420 static void
2421 nfs4close_otw(rnode4_t *rp, cred_t *cred_otw, nfs4_open_owner_t *oop,
2422     nfs4_open_stream_t *osp, int *recov, int *did_start_seqid_syncp,
2423     nfs4_close_type_t close_type, nfs4_error_t *ep, int *have_sync_lockp)
2424 {
2425         COMPOUND4args_clnt args;
2426         COMPOUND4res_clnt res;
2427         CLOSE4args *close_args;
2428         nfs_resop4 *resop;
2429         nfs_argop4 argop[3];
2430         int doqueue = 1;
2431         mntinfo4_t *mi;
2432         seqid4 seqid;
2433         vnode_t *vp;
2434         bool_t needrecov = FALSE;
```



```
2435         nfs4_lost_rqst_t lost_rqst;
2436         hrtime_t t;

2438         ASSERT(nfs_zone() == VTOMI4(RTOV4(rp))->mi_zone);

2440         ASSERT(MUTEX_HELD(&osp->os_sync_lock));

2442         NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE, "nfs4close_otw"));

2444         /* Only set this to 1 if recovery is started */
2445         *recov = 0;

2447         /* do the OTW call to close the file */

2449         if (close_type == CLOSE_RESEND)
2450                 args.ctag = TAG_CLOSE_LOST;
2451         else if (close_type == CLOSE_AFTER_RESEND)
2452                 args.ctag = TAG_CLOSE_UNDO;
2453         else
2454                 args.ctag = TAG_CLOSE;

2456         args.array_len = 3;
2457         args.array = argop;

2459         vp = RTOV4(rp);

2461         mi = VTOMI4(vp);

2463         /* putfh target fh */
2464         argop[0].argop = OP_CPUTFH;
2465         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

2467         argop[1].argop = OP_GETATTR;
2468         argop[1].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
2469         argop[1].nfs_argop4_u.opgetattr.mi = mi;

2471         argop[2].argop = OP_CLOSE;
2472         close_args = &argop[2].nfs_argop4_u.opclose;

2474         seqid = nfs4_get_open_seqid(oop) + 1;

2476         close_args->seqid = seqid;
2477         close_args->open_stateid = osp->open_stateid;

2479         NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
2480             "nfs4close_otw: %s call, rp %s", needrecov ? "recov" : "first",
2481             rnode4info(rp)));

2483         t = gethrtime();

2485         rfs4call(mi, &args, &res, cred_otw, &doqueue, 0, ep);

2487         if (!ep->error && nfs4_need_to_bump_seqid(&res)) {
2488                 nfs4_set_open_seqid(seqid, oop, args.ctag);
2489         }

2491         needrecov = nfs4_needs_recovery(ep, TRUE, mi->mi_vfsp);
2492         if (ep->error && !needrecov) {
2493                 /*
2494                  * if there was an error and no recovery is to be done
2495                  * then then set up the file to flush its cache if
2496                  * needed for the next caller.
2497                  */
2498                 mutex_enter(&rp->r_statelock);
2499                 PURGE_ATTRCACHE4_LOCKED(rp);
2500                 rp->r_flags &= ~R4WRITEMODIFIED;
```

```
2501                mutex_exit(&rp->r_statelock);
2502                return;
2503        }

2505        if (needrecov) {
2506                bool_t abort;
2507                nfs4_bseqid_entry_t *bsep = NULL;

2509                if (close_type != CLOSE_RESEND)
2510                        nfs4close_save_lost_rqst(ep->error, &lost_rqst, oop,
2511                            osp, cred_otw, vp);

2513                if (!ep->error && res.status == NFS4ERR_BAD_SEQID)
2514                        bsep = nfs4_create_bseqid_entry(oop, NULL, vp,
2515                            0, args.ctag, close_args->seqid);

2517                NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
2518                    "nfs4close_otw: initiating recovery. error %d "
2519                    "res.status %d", ep->error, res.status));

2521                /*
2522                 * Drop the 'os_sync_lock' here so we don't hit
2523                 * a potential recursive mutex_enter via an
2524                 * 'open_stream_hold()'.
2525                 */
2526                mutex_exit(&osp->os_sync_lock);
2527                *have_sync_lockp = 0;
2528                abort = nfs4_start_recovery(ep, VTOMI4(vp), vp, NULL, NULL,
2529                    (close_type != CLOSE_RESEND &&
2530                    lost_rqst.lr_op == OP_CLOSE) ? &lost_rqst : NULL,
2531                    OP_CLOSE, bsep, NULL, NULL);

2533                /* drop open seq sync, and let the calling function regrab it */
2534                nfs4_end_open_seqid_sync(oop);
2535                *did_start_seqid_syncp = 0;

2537                if (bsep)
2538                        kmem_free(bsep, sizeof (*bsep));
2539                /*
2540                 * For signals, the caller wants to quit, so don't say to
2541                 * retry.  For forced unmount, if it's a user thread, it
2542                 * wants to quit.  If it's a recovery thread, the retry
2543                 * will happen higher-up on the call stack.  Either way,
2544                 * don't say to retry.
2545                 */
2546                if (abort == FALSE && ep->error != EINTR &&
2547                    !NFS4_FRC_UNMT_ERR(ep->error, mi->mi_vfsp) &&
2548                    close_type != CLOSE_RESEND &&
2549                    close_type != CLOSE_AFTER_RESEND)
2550                        *recov = 1;
2551                else
2552                        *recov = 0;

2554                if (!ep->error)
2555                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2556                return;
2557        }

2559        if (res.status) {
2560                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2561                return;
2562        }

2564        mutex_enter(&rp->r_statev4_lock);
2565        rp->created_v4 = 0;
2566        mutex_exit(&rp->r_statev4_lock);
```



```
2568        resop = &res.array[2];
2569        osp->open_stateid = resop->nfs_resop4_u.opclose.open_stateid;
2570        osp->os_valid = 0;

2572        /*
2573         * This removes the reference obtained at OPEN; ie, when the
2574         * open stream structure was created.
2575         *
2576         * We don't have to worry about calling 'open_stream_rele'
2577         * since we our currently holding a reference to the open
2578         * stream which means the count cannot go to 0 with this
2579         * decrement.
2580         */
2581        ASSERT(osp->os_ref_count >= 2);
2582        osp->os_ref_count--;

2584        if (!ep->error)
2585                nfs4_attr_cache(vp,
2586                    &res.array[1].nfs_resop4_u.opgetattr.ga_res,
2587                    t, cred_otw, TRUE, NULL);

2589        NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE, "nfs4close_otw:"
2590            " returning %d", ep->error));

2592        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
2593 }

2595 /* ARGSUSED */
2596 static int
2597 nfs4_read(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
2598     caller_context_t *ct)
2599 {
2600        rnode4_t *rp;
2601        u_offset_t off;
2602        offset_t diff;
2603        uint_t on;
2604        uint_t n;
2605        caddr_t base;
2606        uint_t flags;
2607        int error;
2608        mntinfo4_t *mi;

2610        rp = VTOR4(vp);

2612        ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));

2614        if (IS_SHADOW(vp, rp))
2615                vp = RTOV4(rp);

2617        if (vp->v_type != VREG)
2618                return (EISDIR);

2620        mi = VTOMI4(vp);

2622        if (nfs_zone() != mi->mi_zone)
2623                return (EIO);

2625        if (uiop->uio_resid == 0)
2626                return (0);

2628        if (uiop->uio_loffset < 0 || uiop->uio_loffset + uiop->uio_resid < 0)
2629                return (EINVAL);

2631        mutex_enter(&rp->r_statelock);
2632        if (rp->r_flags & R4RECOVERRP)
```

```
2633                    error = (rp->r_error ? rp->r_error : EIO);
2634            else
2635                    error = 0;
2636            mutex_exit(&rp->r_statelock);
2637            if (error)
2638                    return (error);

2640            /*
2641             * Bypass VM if caching has been disabled (e.g., locking) or if
2642             * using client-side direct I/O and the file is not mmap'd and
2643             * there are no cached pages.
2644             */
2645            if ((vp->v_flag & VNOCACHE) ||
2646                (((rp->r_flags & R4DIRECTIO) || (mi->mi_flags & MI4_DIRECTIO)) &&
2647                rp->r_mapcnt == 0 && rp->r_inmap == 0 && !nfs4_has_pages(vp))) {
2648                    size_t resid = 0;

2650                    return (nfs4read(vp, NULL, uiop->uio_loffset,
2651                        uiop->uio_resid, &resid, cr, FALSE, uiop));
2652            }

2654            error = 0;

2656            do {
2657                    off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
2658                    on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
2659                    n = MIN(MAXBSIZE - on, uiop->uio_resid);

2661                    if (error = nfs4_validate_caches(vp, cr))
2662                            break;

2664                    mutex_enter(&rp->r_statelock);
2665                    while (rp->r_flags & R4INCACHEPURGE) {
2666                            if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
2667                                    mutex_exit(&rp->r_statelock);
2668                                    return (EINTR);
2669                            }
2670                    }
2671                    diff = rp->r_size - uiop->uio_loffset;
2672                    mutex_exit(&rp->r_statelock);
2673                    if (diff <= 0)
2674                            break;
2675                    if (diff < n)
2676                            n = (uint_t)diff;

2678                    if (vpm_enable) {
2679                            /*
2680                             * Copy data.
2681                             */
2682                            error = vpm_data_copy(vp, off + on, n, uiop,
2683                                1, NULL, 0, S_READ);
2684                    } else {
2685                            base = segmap_getmapflt(segkmap, vp, off + on, n, 1,
2686                                S_READ);

2688                            error = uiomove(base + on, n, UIO_READ, uiop);
2689                    }

2691                    if (!error) {
2692                            /*
2693                             * If read a whole block or read to eof,
2694                             * won't need this buffer again soon.
2695                             */
2696                            mutex_enter(&rp->r_statelock);
2697                            if (n + on == MAXBSIZE ||
2698                                uiop->uio_loffset == rp->r_size)
```



```
2699                                    flags = SM_DONTNEED;
2700                            else
2701                                    flags = 0;
2702                            mutex_exit(&rp->r_statelock);
2703                            if (vpm_enable) {
2704                                    error = vpm_sync_pages(vp, off, n, flags);
2705                            } else {
2706                                    error = segmap_release(segkmap, base, flags);
2707                            }
2708                    } else {
2709                            if (vpm_enable) {
2710                                    (void) vpm_sync_pages(vp, off, n, 0);
2711                            } else {
2712                                    (void) segmap_release(segkmap, base, 0);
2713                            }
2714                    }
2715            } while (!error && uiop->uio_resid > 0);

2717            return (error);
2718    }

2720    /* ARGSUSED */
2721    static int
2722    nfs4_write(vnode_t *vp, struct uio *uiop, int ioflag, cred_t *cr,
2723        caller_context_t *ct)
2724    {
2725            rlim64_t limit = uiop->uio_llimit;
2726            rnode4_t *rp;
2727            u_offset_t off;
2728            caddr_t base;
2729            uint_t flags;
2730            int remainder;
2731            size_t n;
2732            int on;
2733            int error;
2734            int resid;
2735            u_offset_t offset;
2736            mntinfo4_t *mi;
2737            uint_t bsize;

2739            rp = VTOR4(vp);

2741            if (IS_SHADOW(vp, rp))
2742                    vp = RTOV4(rp);

2744            if (vp->v_type != VREG)
2745                    return (EISDIR);

2747            mi = VTOMI4(vp);

2749            if (nfs_zone() != mi->mi_zone)
2750                    return (EIO);

2752            if (uiop->uio_resid == 0)
2753                    return (0);

2755            mutex_enter(&rp->r_statelock);
2756            if (rp->r_flags & R4RECOVERRP)
2757                    error = (rp->r_error ? rp->r_error : EIO);
2758            else
2759                    error = 0;
2760            mutex_exit(&rp->r_statelock);
2761            if (error)
2762                    return (error);

2764            if (ioflag & FAPPEND) {
```

```
2765                struct vattr va;

2767                /*
2768                 * Must serialize if appending.
2769                 */
2770                if (nfs_rw_lock_held(&rp->r_rwlock, RW_READER)) {
2771                        nfs_rw_exit(&rp->r_rwlock);
2772                        if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER,
2773                            INTR4(vp)))
2774                                return (EINTR);
2775                }

2777                va.va_mask = AT_SIZE;
2778                error = nfs4getattr(vp, &va, cr);
2779                if (error)
2780                        return (error);
2781                uiop->uio_loffset = va.va_size;
2782        }

2784        offset = uiop->uio_loffset + uiop->uio_resid;

2786        if (uiop->uio_loffset < (offset_t)0 || offset < 0)
2787                return (EINVAL);

2789        if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
2790                limit = MAXOFFSET_T;

2792        /*
2793         * Check to make sure that the process will not exceed
2794         * its limit on file size.  It is okay to write up to
2795         * the limit, but not beyond.  Thus, the write which
2796         * reaches the limit will be short and the next write
2797         * will return an error.
2798         */
2799        remainder = 0;
2800        if (offset > uiop->uio_llimit) {
2801                remainder = offset - uiop->uio_llimit;
2802                uiop->uio_resid = uiop->uio_llimit - uiop->uio_loffset;
2803                if (uiop->uio_resid <= 0) {
2804                        proc_t *p = ttoproc(curthread);

2806                        uiop->uio_resid += remainder;
2807                        mutex_enter(&p->p_lock);
2808                        (void) rctl_action(rctlproc_legacy[RLIMIT_FSIZE],
2809                            p->p_rctls, p, RCA_UNSAFE_SIGINFO);
2810                        mutex_exit(&p->p_lock);
2811                        return (EFBIG);
2812                }
2813        }

2815        /* update the change attribute, if we have a write delegation */

2817        mutex_enter(&rp->r_statev4_lock);
2818        if (rp->r_deleg_type == OPEN_DELEGATE_WRITE)
2819                rp->r_deleg_change++;

2821        mutex_exit(&rp->r_statev4_lock);

2823        if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR4(vp)))
2824                return (EINTR);

2826        /*
2827         * Bypass VM if caching has been disabled (e.g., locking) or if
2828         * using client-side direct I/O and the file is not mmap'd and
2829         * there are no cached pages.
2830         */
```



```
2831        if ((vp->v_flag & VNOCACHE) ||
2832            (((rp->r_flags & R4DIRECTIO) || (mi->mi_flags & MI4_DIRECTIO)) &&
2833            rp->r_mapcnt == 0 && rp->r_inmap == 0 && !nfs4_has_pages(vp))) {
2834                size_t bufsize;
2835                int count;
2836                u_offset_t org_offset;
2837                stable_how4 stab_comm;
2838 nfs4_fwrite:
2839                if (rp->r_flags & R4STALE) {
2840                        resid = uiop->uio_resid;
2841                        offset = uiop->uio_loffset;
2842                        error = rp->r_error;
2843                        /*
2844                         * A close may have cleared r_error, if so,
2845                         * propagate ESTALE error return properly
2846                         */
2847                        if (error == 0)
2848                                error = ESTALE;
2849                        goto bottom;
2850                }

2852                bufsize = MIN(uiop->uio_resid, mi->mi_stsize);
2853                base = kmem_alloc(bufsize, KM_SLEEP);
2854                do {
2855                        if (ioflag & FDSYNC)
2856                                stab_comm = DATA_SYNC4;
2857                        else
2858                                stab_comm = FILE_SYNC4;
2859                        resid = uiop->uio_resid;
2860                        offset = uiop->uio_loffset;
2861                        count = MIN(uiop->uio_resid, bufsize);
2862                        org_offset = uiop->uio_loffset;
2863                        error = uiomove(base, count, UIO_WRITE, uiop);
2864                        if (!error) {
2865                                error = nfs4write(vp, base, org_offset,
2866                                    count, cr, &stab_comm);
2867                                if (!error) {
2868                                        mutex_enter(&rp->r_statelock);
2869                                        if (rp->r_size < uiop->uio_loffset)
2870                                                rp->r_size = uiop->uio_loffset;
2871                                        mutex_exit(&rp->r_statelock);
2872                                }
2873                        }
2874                } while (!error && uiop->uio_resid > 0);
2875                kmem_free(base, bufsize);
2876                goto bottom;
2877        }

2879        bsize = vp->v_vfsp->vfs_bsize;

2881        do {
2882                off = uiop->uio_loffset & MAXBMASK; /* mapping offset */
2883                on = uiop->uio_loffset & MAXBOFFSET; /* Relative offset */
2884                n = MIN(MAXBSIZE - on, uiop->uio_resid);

2886                resid = uiop->uio_resid;
2887                offset = uiop->uio_loffset;

2889                if (rp->r_flags & R4STALE) {
2890                        error = rp->r_error;
2891                        /*
2892                         * A close may have cleared r_error, if so,
2893                         * propagate ESTALE error return properly
2894                         */
2895                        if (error == 0)
2896                                error = ESTALE;
```

```
2897                    break;
2898            }

2900            /*
2901             * Don't create dirty pages faster than they
2902             * can be cleaned so that the system doesn't
2903             * get imbalanced.  If the async queue is
2904             * maxed out, then wait for it to drain before
2905             * creating more dirty pages.  Also, wait for
2906             * any threads doing pagewalks in the vop_getattr
2907             * entry points so that they don't block for
2908             * long periods.
2909             */
2910            mutex_enter(&rp->r_statelock);
2911            while ((mi->mi_max_threads != 0 &&
2912                rp->r_awcount > 2 * mi->mi_max_threads) ||
2913                rp->r_gcount > 0) {
2914                if (INTR4(vp)) {
2915                        klwp_t *lwp = ttolwp(curthread);

2917                        if (lwp != NULL)
2918                                lwp->lwp_nostop++;
2919                        if (!cv_wait_sig(&rp->r_cv, &rp->r_statelock)) {
2920                                mutex_exit(&rp->r_statelock);
2921                                if (lwp != NULL)
2922                                        lwp->lwp_nostop--;
2923                                error = EINTR;
2924                                goto bottom;
2925                        }
2926                        if (lwp != NULL)
2927                                lwp->lwp_nostop--;
2928                } else
2929                        cv_wait(&rp->r_cv, &rp->r_statelock);
2930            }
2931            mutex_exit(&rp->r_statelock);

2933            /*
2934             * Touch the page and fault it in if it is not in core
2935             * before segmap_getmapflt or vpm_data_copy can lock it.
2936             * This is to avoid the deadlock if the buffer is mapped
2937             * to the same file through mmap which we want to write.
2938             */
2939            uio_prefaultpages((long)n, uiop);

2941            if (vpm_enable) {
2942                    /*
2943                     * It will use kpm mappings, so no need to
2944                     * pass an address.
2945                     */
2946                    error = writerp4(rp, NULL, n, uiop, 0);
2947            } else  {
2948                    if (segmap_kpm) {
2949                            int pon = uiop->uio_loffset & PAGEOFFSET;
2950                            size_t pn = MIN(PAGESIZE - pon,
2951                                uiop->uio_resid);
2952                            int pagecreate;

2954                            mutex_enter(&rp->r_statelock);
2955                            pagecreate = (pon == 0) && (pn == PAGESIZE ||
2956                                uiop->uio_loffset + pn >= rp->r_size);
2957                            mutex_exit(&rp->r_statelock);

2959                            base = segmap_getmapflt(segkmap, vp, off + on,
2960                                pn, !pagecreate, S_WRITE);

2962                            error = writerp4(rp, base + pon, n, uiop,
```



```
2963                                    pagecreate);

2965                            } else {
2966                                    base = segmap_getmapflt(segkmap, vp, off + on,
2967                                        n, 0, S_READ);
2968                                    error = writerp4(rp, base + on, n, uiop, 0);
2969                            }
2970                    }

2972                    if (!error) {
2973                            if (mi->mi_flags & MI4_NOAC)
2974                                    flags = SM_WRITE;
2975                            else if ((uiop->uio_loffset % bsize) == 0 ||
2976                                IS_SWAPVP(vp)) {
2977                                    /*
2978                                     * Have written a whole block.
2979                                     * Start an asynchronous write
2980                                     * and mark the buffer to
2981                                     * indicate that it won't be
2982                                     * needed again soon.
2983                                     */
2984                                    flags = SM_WRITE | SM_ASYNC | SM_DONTNEED;
2985                            } else
2986                                    flags = 0;
2987                            if ((ioflag & (FSYNC|FDSYNC)) ||
2988                                (rp->r_flags & R4OUTOFSPACE)) {
2989                                    flags &= ~SM_ASYNC;
2990                                    flags |= SM_WRITE;
2991                            }
2992                            if (vpm_enable) {
2993                                    error = vpm_sync_pages(vp, off, n, flags);
2994                            } else {
2995                                    error = segmap_release(segkmap, base, flags);
2996                            }
2997                    } else {
2998                            if (vpm_enable) {
2999                                    (void) vpm_sync_pages(vp, off, n, 0);
3000                            } else {
3001                                    (void) segmap_release(segkmap, base, 0);
3002                            }
3003                            /*
3004                             * In the event that we got an access error while
3005                             * faulting in a page for a write-only file just
3006                             * force a write.
3007                             */
3008                            if (error == EACCES)
3009                                    goto nfs4_fwrite;
3010                    }
3011            } while (!error && uiop->uio_resid > 0);

3013    bottom:
3014            if (error) {
3015                    uiop->uio_resid = resid + remainder;
3016                    uiop->uio_loffset = offset;
3017            } else {
3018                    uiop->uio_resid += remainder;

3020                    mutex_enter(&rp->r_statev4_lock);
3021                    if (rp->r_deleg_type == OPEN_DELEGATE_WRITE) {
3022                            gethrestime(&rp->r_attr.va_mtime);
3023                            rp->r_attr.va_ctime = rp->r_attr.va_mtime;
3024                    }
3025                    mutex_exit(&rp->r_statev4_lock);
3026            }

3028            nfs_rw_exit(&rp->r_lkserlock);
```

```
3030         return (error);
3031 }

3033 /*
3034  * Flags are composed of {B_ASYNC, B_INVAL, B_FREE, B_DONTNEED}
3035  */
3036 static int
3037 nfs4_rdwrlbn(vnode_t *vp, page_t *pp, u_offset_t off, size_t len,
3038     int flags, cred_t *cr)
3039 {
3040         struct buf *bp;
3041         int error;
3042         page_t *savepp;
3043         uchar_t fsdata;
3044         stable_how4 stab_comm;

3046         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
3047         bp = pageio_setup(pp, len, vp, flags);
3048         ASSERT(bp != NULL);

3050         /*
3051          * pageio_setup should have set b_addr to 0.  This
3052          * is correct since we want to do I/O on a page
3053          * boundary.  bp_mapin will use this addr to calculate
3054          * an offset, and then set b_addr to the kernel virtual
3055          * address it allocated for us.
3056          */
3057         ASSERT(bp->b_un.b_addr == 0);

3059         bp->b_edev = 0;
3060         bp->b_dev = 0;
3061         bp->b_lblkno = lbtodb(off);
3062         bp->b_file = vp;
3063         bp->b_offset = (offset_t)off;
3064         bp_mapin(bp);

3066         if ((flags & (B_WRITE|B_ASYNC)) == (B_WRITE|B_ASYNC) &&
3067             freemem > desfree)
3068                 stab_comm = UNSTABLE4;
3069         else
3070                 stab_comm = FILE_SYNC4;

3072         error = nfs4_bio(bp, &stab_comm, cr, FALSE);

3074         bp_mapout(bp);
3075         pageio_done(bp);

3077         if (stab_comm == UNSTABLE4)
3078                 fsdata = C_DELAYCOMMIT;
3079         else
3080                 fsdata = C_NOCOMMIT;

3082         savepp = pp;
3083         do {
3084                 pp->p_fsdata = fsdata;
3085         } while ((pp = pp->p_next) != savepp);

3087         return (error);
3088 }

3090 /*
3091  */
3092 static int
3093 nfs4rdwr_check_osid(vnode_t *vp, nfs4_error_t *ep, cred_t *cr)
3094 {
```



```
3095         nfs4_open_owner_t       *oop;
3096         nfs4_open_stream_t      *osp;
3097         rnode4_t                *rp = VTOR4(vp);
3098         mntinfo4_t              *mi = VTOMI4(vp);
3099         int                     reopen_needed;

3101         ASSERT(nfs_zone() == mi->mi_zone);


3104         oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
3105         if (!oop)
3106                 return (EIO);

3108         /* returns with 'os_sync_lock' held */
3109         osp = find_open_stream(oop, rp);
3110         if (!osp) {
3111                 open_owner_rele(oop);
3112                 return (EIO);
3113         }

3115         if (osp->os_failed_reopen) {
3116                 mutex_exit(&osp->os_sync_lock);
3117                 open_stream_rele(osp, rp);
3118                 open_owner_rele(oop);
3119                 return (EIO);
3120         }

3122         /*
3123          * Determine whether a reopen is needed.  If this
3124          * is a delegation open stream, then the os_delegation bit
3125          * should be set.
3126          */

3128         reopen_needed = osp->os_delegation;

3130         mutex_exit(&osp->os_sync_lock);
3131         open_owner_rele(oop);

3133         if (reopen_needed) {
3134                 nfs4_error_zinit(ep);
3135                 nfs4_reopen(vp, osp, ep, CLAIM_NULL, FALSE, FALSE);
3136                 mutex_enter(&osp->os_sync_lock);
3137                 if (ep->error || ep->stat || osp->os_failed_reopen) {
3138                         mutex_exit(&osp->os_sync_lock);
3139                         open_stream_rele(osp, rp);
3140                         return (EIO);
3141                 }
3142                 mutex_exit(&osp->os_sync_lock);
3143         }
3144         open_stream_rele(osp, rp);

3146         return (0);
3147 }

3149 /*
3150  * Write to file.  Writes to remote server in largest size
3151  * chunks that the server can handle.  Write is synchronous.
3152  */
3153 static int
3154 nfs4write(vnode_t *vp, caddr_t base, u_offset_t offset, int count, cred_t *cr,
3155     stable_how4 *stab_comm)
3156 {
3157         mntinfo4_t *mi;
3158         COMPOUND4args_clnt args;
3159         COMPOUND4res_clnt res;
3160         WRITE4args *wargs;
```

```
3161         WRITE4res *wres;
3162         nfs_argop4 argop[2];
3163         nfs_resop4 *resop;
3164         int tsize;
3165         stable_how4 stable;
3166         rnode4_t *rp;
3167         int doqueue = 1;
3168         bool_t needrecov;
3169         nfs4_recov_state_t recov_state;
3170         nfs4_stateid_types_t sid_types;
3171         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
3172         int recov;

3174         rp = VTOR4(vp);
3175         mi = VTOMI4(vp);

3177         ASSERT(nfs_zone() == mi->mi_zone);

3179         stable = *stab_comm;
3180         *stab_comm = FILE_SYNC4;

3182         needrecov = FALSE;
3183         recov_state.rs_flags = 0;
3184         recov_state.rs_num_retry_despite_err = 0;
3185         nfs4_init_stateid_types(&sid_types);

3187         /* Is curthread the recovery thread? */
3188         mutex_enter(&mi->mi_lock);
3189         recov = (mi->mi_recovthread == curthread);
3190         mutex_exit(&mi->mi_lock);

3192 recov_retry:
3193         args.ctag = TAG_WRITE;
3194         args.array_len = 2;
3195         args.array = argop;

3197         if (!recov) {
3198                 e.error = nfs4_start_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3199                     &recov_state, NULL);
3200                 if (e.error)
3201                         return (e.error);
3202         }

3204         /* 0. putfh target fh */
3205         argop[0].argop = OP_CPUTFH;
3206         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

3208         /* 1. write */
3209         nfs4args_write(&argop[1], stable, rp, cr, &wargs, &sid_types);

3211         do {

3213                 wargs->offset = (offset4)offset;
3214                 wargs->data_val = base;

3216                 if (mi->mi_io_kstats) {
3217                         mutex_enter(&mi->mi_lock);
3218                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
3219                         mutex_exit(&mi->mi_lock);
3220                 }

3222                 if ((vp->v_flag & VNOCACHE) ||
3223                     (rp->r_flags & R4DIRECTIO) ||
3224                     (mi->mi_flags & MI4_DIRECTIO))
3225                         tsize = MIN(mi->mi_stsize, count);
3226                 else
```



```
3227                         tsize = MIN(mi->mi_curwrite, count);
3228                 wargs->data_len = (uint_t)tsize;
3229                 rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

3231                 if (mi->mi_io_kstats) {
3232                         mutex_enter(&mi->mi_lock);
3233                         kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
3234                         mutex_exit(&mi->mi_lock);
3235                 }

3237                 if (!recov) {
3238                         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
3239                         if (e.error && !needrecov) {
3240                                 nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3241                                     &recov_state, needrecov);
3242                                 return (e.error);
3243                         }
3244                 } else {
3245                         if (e.error)
3246                                 return (e.error);
3247                 }

3249                 /*
3250                  * Do handling of OLD_STATEID outside
3251                  * of the normal recovery framework.
3252                  *
3253                  * If write receives a BAD stateid error while using a
3254                  * delegation stateid, retry using the open stateid (if it
3255                  * exists).  If it doesn't have an open stateid, reopen the
3256                  * file first, then retry.
3257                  */
3258                 if (!e.error && res.status == NFS4ERR_OLD_STATEID &&
3259                     sid_types.cur_sid_type != SPEC_SID) {
3260                         nfs4_save_stateid(&wargs->stateid, &sid_types);
3261                         if (!recov)
3262                                 nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3263                                     &recov_state, needrecov);
3264                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3265                         goto recov_retry;
3266                 } else if (e.error == 0 && res.status == NFS4ERR_BAD_STATEID &&
3267                     sid_types.cur_sid_type == DEL_SID) {
3268                         nfs4_save_stateid(&wargs->stateid, &sid_types);
3269                         mutex_enter(&rp->r_statev4_lock);
3270                         rp->r_deleg_return_pending = TRUE;
3271                         mutex_exit(&rp->r_statev4_lock);
3272                         if (nfs4rdwr_check_osid(vp, &e, cr)) {
3273                                 if (!recov)
3274                                         nfs4_end_fop(mi, vp, NULL, OH_WRITE,
3275                                             &recov_state, needrecov);
3276                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3277                                     (caddr_t)&res);
3278                                 return (EIO);
3279                         }
3280                         if (!recov)
3281                                 nfs4_end_fop(mi, vp, NULL, OH_WRITE,
3282                                     &recov_state, needrecov);
3283                         /* hold needed for nfs4delegreturn_thread */
3284                         VN_HOLD(vp);
3285                         nfs4delegreturn_async(rp, (NFS4_DR_PUSH|NFS4_DR_REOPEN|
3286                             NFS4_DR_DISCARD), FALSE);
3287                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3288                         goto recov_retry;
3289                 }

3291                 if (needrecov) {
3292                         bool_t abort;
```

```
3294                    NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
3295                        "nfs4write: client got error %d, res.status %d"
3296                        ", so start recovery", e.error, res.status));
3297
3298                    abort = nfs4_start_recovery(&e,
3299                        VTOMI4(vp), vp, NULL, &wargs->stateid,
3300                        NULL, OP_WRITE, NULL, NULL, NULL);
3301                    if (!e.error) {
3302                        e.error = geterrno4(res.status);
3303                        (void) xdr_free(xdr_COMPOUND4res_clnt,
3304                            (caddr_t)&res);
3305                    }
3306                    nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3307                        &recov_state, needrecov);
3308                    if (abort == FALSE)
3309                        goto recov_retry;
3310                    return (e.error);
3311                }
3312
3313                if (res.status) {
3314                    e.error = geterrno4(res.status);
3315                    (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3316                    if (!recov)
3317                        nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3318                            &recov_state, needrecov);
3319                    return (e.error);
3320                }
3321
3322                resop = &res.array[1];  /* write res */
3323                wres = &resop->nfs_resop4_u.opwrite;
3324
3325                if ((int)wres->count > tsize) {
3326                    (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3327
3328                    zcmn_err(getzoneid(), CE_WARN,
3329                        "nfs4write: server wrote %u, requested was %u",
3330                        (int)wres->count, tsize);
3331                    if (!recov)
3332                        nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE,
3333                            &recov_state, needrecov);
3334                    return (EIO);
3335                }
3336                if (wres->committed == UNSTABLE4) {
3337                    *stab_comm = UNSTABLE4;
3338                    if (wargs->stable == DATA_SYNC4 ||
3339                        wargs->stable == FILE_SYNC4) {
3340                        (void) xdr_free(xdr_COMPOUND4res_clnt,
3341                            (caddr_t)&res);
3342                        zcmn_err(getzoneid(), CE_WARN,
3343                            "nfs4write: server %s did not commit "
3344                            "to stable storage",
3345                            rp->r_server->sv_hostname);
3346                        if (!recov)
3347                            nfs4_end_fop(VTOMI4(vp), vp, NULL,
3348                                OH_WRITE, &recov_state, needrecov);
3349                        return (EIO);
3350                    }
3351                }
3352
3353                tsize = (int)wres->count;
3354                count -= tsize;
3355                base += tsize;
3356                offset += tsize;
3357                if (mi->mi_io_kstats) {
3358                    mutex_enter(&mi->mi_lock);
```



```
3359                        KSTAT_IO_PTR(mi->mi_io_kstats)->writes++;
3360                        KSTAT_IO_PTR(mi->mi_io_kstats)->nwritten +=
3361                            tsize;
3362                        mutex_exit(&mi->mi_lock);
3363                }
3364                lwp_stat_update(LWP_STAT_OUBLK, 1);
3365                mutex_enter(&rp->r_statelock);
3366                if (rp->r_flags & R4HAVEVERF) {
3367                        if (rp->r_writeverf != wres->writeverf) {
3368                                nfs4_set_mod(vp);
3369                                rp->r_writeverf = wres->writeverf;
3370                        }
3371                } else {
3372                        rp->r_writeverf = wres->writeverf;
3373                        rp->r_flags |= R4HAVEVERF;
3374                }
3375                PURGE_ATTRCACHE4_LOCKED(rp);
3376                rp->r_flags |= R4WRITEMODIFIED;
3377                gethrestime(&rp->r_attr.va_mtime);
3378                rp->r_attr.va_ctime = rp->r_attr.va_mtime;
3379                mutex_exit(&rp->r_statelock);
3380                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3381        } while (count);
3382
3383        if (!recov)
3384                nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_WRITE, &recov_state,
3385                    needrecov);
3386
3387        return (e.error);
3388 }
3389
3390 /*
3391  * Read from a file.  Reads data in largest chunks our interface can handle.
3392  */
3393 static int
3394 nfs4read(vnode_t *vp, caddr_t base, offset_t offset, int count,
3395     size_t *residp, cred_t *cr, bool_t async, struct uio *uiop)
3396 {
3397        mntinfo4_t *mi;
3398        COMPOUND4args_clnt args;
3399        COMPOUND4res_clnt res;
3400        READ4args *rargs;
3401        nfs_argop4 argop[2];
3402        int tsize;
3403        int doqueue;
3404        rnode4_t *rp;
3405        int data_len;
3406        bool_t is_eof;
3407        bool_t needrecov = FALSE;
3408        nfs4_recov_state_t recov_state;
3409        nfs4_stateid_types_t sid_types;
3410        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
3411
3412        rp = VTOR4(vp);
3413        mi = VTOMI4(vp);
3414        doqueue = 1;
3415
3416        ASSERT(nfs_zone() == mi->mi_zone);
3417
3418        args.ctag = async ? TAG_READAHEAD : TAG_READ;
3419
3420        args.array_len = 2;
3421        args.array = argop;
3422
3423        nfs4_init_stateid_types(&sid_types);
```

```
3425         recov_state.rs_flags = 0;
3426         recov_state.rs_num_retry_despite_err = 0;

3428 recov_retry:
3429         e.error = nfs4_start_fop(mi, vp, NULL, OH_READ,
3430             &recov_state, NULL);
3431         if (e.error)
3432                 return (e.error);

3434         /* putfh target fh */
3435         argop[0].argop = OP_CPUTFH;
3436         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

3438         /* read */
3439         argop[1].argop = OP_READ;
3440         rargs = &argop[1].nfs_argop4_u.opread;
3441         rargs->stateid = nfs4_get_stateid(cr, rp, curproc->p_pidp->pid_id, mi,
3442             OP_READ, &sid_types, async);

3444         do {
3445                 if (mi->mi_io_kstats) {
3446                         mutex_enter(&mi->mi_lock);
3447                         kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
3448                         mutex_exit(&mi->mi_lock);
3449                 }

3451                 NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
3452                     "nfs4read: %s call, rp %s",
3453                     needrecov ? "recov" : "first",
3454                     rnode4info(rp)));

3456                 if ((vp->v_flag & VNOCACHE) ||
3457                     (rp->r_flags & R4DIRECTIO) ||
3458                     (mi->mi_flags & MI4_DIRECTIO))
3459                         tsize = MIN(mi->mi_tsize, count);
3460                 else
3461                         tsize = MIN(mi->mi_curread, count);

3463                 rargs->offset = (offset4)offset;
3464                 rargs->count = (count4)tsize;
3465                 rargs->res_data_val_alt = NULL;
3466                 rargs->res_mblk = NULL;
3467                 rargs->res_uiop = NULL;
3468                 rargs->res_maxsize = 0;
3469                 rargs->wlist = NULL;

3471                 if (uiop)
3472                         rargs->res_uiop = uiop;
3473                 else
3474                         rargs->res_data_val_alt = base;
3475                 rargs->res_maxsize = tsize;

3477                 rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);
3478 #ifdef  DEBUG
3479                 if (nfs4read_error_inject) {
3480                         res.status = nfs4read_error_inject;
3481                         nfs4read_error_inject = 0;
3482                 }
3483 #endif

3485                 if (mi->mi_io_kstats) {
3486                         mutex_enter(&mi->mi_lock);
3487                         kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
3488                         mutex_exit(&mi->mi_lock);
3489                 }
```



```
3491                 needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
3492                 if (e.error != 0 && !needrecov) {
3493                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3494                             &recov_state, needrecov);
3495                         return (e.error);
3496                 }

3498                 /*
3499                  * Do proper retry for OLD and BAD stateid errors outside
3500                  * of the normal recovery framework.  There are two differences
3501                  * between async and sync reads.  The first is that we allow
3502                  * retry on BAD_STATEID for async reads, but not sync reads.
3503                  * The second is that we mark the file dead for a failed
3504                  * attempt with a special stateid for sync reads, but just
3505                  * return EIO for async reads.
3506                  *
3507                  * If a sync read receives a BAD stateid error while using a
3508                  * delegation stateid, retry using the open stateid (if it
3509                  * exists).  If it doesn't have an open stateid, reopen the
3510                  * file first, then retry.
3511                  */
3512                 if (e.error == 0 && (res.status == NFS4ERR_OLD_STATEID ||
3513                     res.status == NFS4ERR_BAD_STATEID) && async) {
3514                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3515                             &recov_state, needrecov);
3516                         if (sid_types.cur_sid_type == SPEC_SID) {
3517                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3518                                     (caddr_t)&res);
3519                                 return (EIO);
3520                         }
3521                         nfs4_save_stateid(&rargs->stateid, &sid_types);
3522                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3523                         goto recov_retry;
3524                 } else if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
3525                     !async && sid_types.cur_sid_type != SPEC_SID) {
3526                         nfs4_save_stateid(&rargs->stateid, &sid_types);
3527                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3528                             &recov_state, needrecov);
3529                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3530                         goto recov_retry;
3531                 } else if (e.error == 0 && res.status == NFS4ERR_BAD_STATEID &&
3532                     sid_types.cur_sid_type == DEL_SID) {
3533                         nfs4_save_stateid(&rargs->stateid, &sid_types);
3534                         mutex_enter(&rp->r_statev4_lock);
3535                         rp->r_deleg_return_pending = TRUE;
3536                         mutex_exit(&rp->r_statev4_lock);
3537                         if (nfs4rdwr_check_osid(vp, &e, cr)) {
3538                                 nfs4_end_fop(mi, vp, NULL, OH_READ,
3539                                     &recov_state, needrecov);
3540                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
3541                                     (caddr_t)&res);
3542                                 return (EIO);
3543                         }
3544                         nfs4_end_fop(mi, vp, NULL, OH_READ,
3545                             &recov_state, needrecov);
3546                         /* hold needed for nfs4delegreturn_thread */
3547                         VN_HOLD(vp);
3548                         nfs4delegreturn_async(rp, (NFS4_DR_PUSH|NFS4_DR_REOPEN|
3549                             NFS4_DR_DISCARD), FALSE);
3550                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3551                         goto recov_retry;
3552                 }
3553                 if (needrecov) {
3554                         bool_t abort;

3556                         NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
```

```
3557                            "nfs4read: initiating recovery\n"));
3558                        abort = nfs4_start_recovery(&e,
3559                            mi, vp, NULL, &rargs->stateid,
3560                            NULL, OP_READ, NULL, NULL, NULL);
3561                        nfs4_end_fop(mi, vp, NULL, OH_READ,
3562                            &recov_state, needrecov);
3563                        /*
3564                         * Do not retry if we got OLD_STATEID using a special
3565                         * stateid.  This avoids looping with a broken server.
3566                         */
3567                        if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
3568                            sid_types.cur_sid_type == SPEC_SID)
3569                                abort = TRUE;

3571                        if (abort == FALSE) {
3572                                /*
3573                                 * Need to retry all possible stateids in
3574                                 * case the recovery error wasn't stateid
3575                                 * related or the stateids have become
3576                                 * stale (server reboot).
3577                                 */
3578                                nfs4_init_stateid_types(&sid_types);
3579                                (void) xdr_free(xdr_COMPOUND4res_clnt,
3580                                    (caddr_t)&res);
3581                                goto recov_retry;
3582                        }

3584                        if (!e.error) {
3585                                e.error = geterrno4(res.status);
3586                                (void) xdr_free(xdr_COMPOUND4res_clnt,
3587                                    (caddr_t)&res);
3588                        }
3589                        return (e.error);
3590                }

3592                if (res.status) {
3593                        e.error = geterrno4(res.status);
3594                        nfs4_end_fop(mi, vp, NULL, OH_READ,
3595                            &recov_state, needrecov);
3596                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
3597                        return (e.error);
3598                }

3600                data_len = res.array[1].nfs_resop4_u.opread.data_len;
3601                count -= data_len;
3602                if (base)
3603                        base += data_len;
3604                offset += data_len;
3605                if (mi->mi_io_kstats) {
3606                        mutex_enter(&mi->mi_lock);
3607                        KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
3608                        KSTAT_IO_PTR(mi->mi_io_kstats)->nread += data_len;
3609                        mutex_exit(&mi->mi_lock);
3610                }
3611                lwp_stat_update(LWP_STAT_INBLK, 1);
3612                is_eof = res.array[1].nfs_resop4_u.opread.eof;
3613                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

3615        } while (count && !is_eof);

3617        *residp = count;

3619        nfs4_end_fop(mi, vp, NULL, OH_READ, &recov_state, needrecov);

3621        return (e.error);
3622 }
```



```
3624 /* ARGSUSED */
3625 static int
3626 nfs4_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag, cred_t *cr, int *rvalp,
3627         caller_context_t *ct)
3628 {
3629         if (nfs_zone() != VTOMI4(vp)->mi_zone)
3630                 return (EIO);
3631         switch (cmd) {
3632                 case _FIODIRECTIO:
3633                         return (nfs4_directio(vp, (int)arg, cr));
3634                 default:
3635                         return (ENOTTY);
3636         }
3637 }

3639 /* ARGSUSED */
3640 int
3641 nfs4_getattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
3642     caller_context_t *ct)
3643 {
3644         int error;
3645         rnode4_t *rp = VTOR4(vp);

3647         if (nfs_zone() != VTOMI4(vp)->mi_zone)
3648                 return (EIO);
3649         /*
3650          * If it has been specified that the return value will
3651          * just be used as a hint, and we are only being asked
3652          * for size, fsid or rdevid, then return the client's
3653          * notion of these values without checking to make sure
3654          * that the attribute cache is up to date.
3655          * The whole point is to avoid an over the wire GETATTR
3656          * call.
3657          */
3658         if (flags & ATTR_HINT) {
3659                 if (!(vap->va_mask & ~(AT_SIZE | AT_FSID | AT_RDEV))) {
3660                         mutex_enter(&rp->r_statelock);
3661                         if (vap->va_mask & AT_SIZE)
3662                                 vap->va_size = rp->r_size;
3663                         if (vap->va_mask & AT_FSID)
3664                                 vap->va_fsid = rp->r_attr.va_fsid;
3665                         if (vap->va_mask & AT_RDEV)
3666                                 vap->va_rdev = rp->r_attr.va_rdev;
3667                         mutex_exit(&rp->r_statelock);
3668                         return (0);
3669                 }
3670         }

3672         /*
3673          * Only need to flush pages if asking for the mtime
3674          * and if there any dirty pages or any outstanding
3675          * asynchronous (write) requests for this file.
3676          */
3677         if (vap->va_mask & AT_MTIME) {
3678                 rp = VTOR4(vp);
3679                 if (nfs4_has_pages(vp)) {
3680                         mutex_enter(&rp->r_statev4_lock);
3681                         if (rp->r_deleg_type != OPEN_DELEGATE_WRITE) {
3682                                 mutex_exit(&rp->r_statev4_lock);
3683                                 if (rp->r_flags & R4DIRTY ||
3684                                     rp->r_awcount > 0) {
3685                                         mutex_enter(&rp->r_statelock);
3686                                         rp->r_gcount++;
3687                                         mutex_exit(&rp->r_statelock);
3688                                         error =
```

```
3689                                            nfs4_putpage(vp, (u_offset_t)0,
3690                                            0, 0, cr, NULL);
3691                                    mutex_enter(&rp->r_statelock);
3692                                    if (error && (error == ENOSPC ||
3693                                        error == EDQUOT)) {
3694                                            if (!rp->r_error)
3695                                                    rp->r_error = error;
3696                                    }
3697                                    if (--rp->r_gcount == 0)
3698                                            cv_broadcast(&rp->r_cv);
3699                                    mutex_exit(&rp->r_statelock);
3700                            }
3701                    } else {
3702                            mutex_exit(&rp->r_statev4_lock);
3703                    }
3704            }
3705        }
3706        return (nfs4getattr(vp, vap, cr));
3707 }

3709 int
3710 nfs4_compare_modes(mode_t from_server, mode_t on_client)
3711 {
3712        /*
3713         * If these are the only two bits cleared
3714         * on the server then return 0 (OK) else
3715         * return 1 (BAD).
3716         */
3717        on_client &= ~(S_ISUID|S_ISGID);
3718        if (on_client == from_server)
3719                return (0);
3720        else
3721                return (1);
3722 }

3724 /*ARGSUSED4*/
3725 static int
3726 nfs4_setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
3727     caller_context_t *ct)
3728 {
3729        int error;

3731        if (vap->va_mask & AT_NOSET)
3732                return (EINVAL);

3734        if (nfs_zone() != VTOMI4(vp)->mi_zone)
3735                return (EIO);

3737        /*
3738         * Don't call secpolicy_vnode_setattr, the client cannot
3739         * use its cached attributes to make security decisions
3740         * as the server may be faking mode bits or mapping uid/gid.
3741         * Always just let the server to the checking.
3742         * If we provide the ability to remove basic priviledges
3743         * to setattr (e.g. basic without chmod) then we will
3744         * need to add a check here before calling the server.
3745         */
3746        error = nfs4setattr(vp, vap, flags, cr, NULL);

3748        if (error == 0 && (vap->va_mask & AT_SIZE) && vap->va_size == 0)
3749                vnevent_truncate(vp, ct);

3751        return (error);
3752 }

3754 /*
```



```
3755  * To replace the "guarded" version 3 setattr, we use two types of compound
3756  * setattr requests:
3757  * 1. The "normal" setattr, used when the size of the file isn't being
3758  *    changed - { Putfh <fh>; Setattr; Getattr }/
3759  * 2. If the size is changed, precede Setattr with: Getattr; Verify
3760  *    with only ctime as the argument. If the server ctime differs from
3761  *    what is cached on the client, the verify will fail, but we would
3762  *    already have the ctime from the preceding getattr, so just set it
3763  *    and retry. Thus the compound here is - { Putfh <fh>; Getattr; Verify;
3764  *      Setattr; Getattr }.
3765  *
3766  * The vsecattr_t * input parameter will be non-NULL if ACLs are being set in
3767  * this setattr and NULL if they are not.
3768  */
3769 static int
3770 nfs4setattr(vnode_t *vp, struct vattr *vap, int flags, cred_t *cr,
3771     vsecattr_t *vsap)
3772 {
3773        COMPOUND4args_clnt args;
3774        COMPOUND4res_clnt res, *resp = NULL;
3775        nfs4_ga_res_t *garp = NULL;
3776        int numops = 3;                 /* { Putfh; Setattr; Getattr } */
3777        nfs_argop4 argop[5];
3778        int verify_argop = -1;
3779        int setattr_argop = 1;
3780        nfs_resop4 *resop;
3781        vattr_t va;
3782        rnode4_t *rp;
3783        int doqueue = 1;
3784        uint_t mask = vap->va_mask;
3785        mode_t omode;
3786        vsecattr_t *vsp;
3787        timestruc_t ctime;
3788        bool_t needrecov = FALSE;
3789        nfs4_recov_state_t recov_state;
3790        nfs4_stateid_types_t sid_types;
3791        stateid4 stateid;
3792        hrtime_t t;
3793        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
3794        servinfo4_t *svp;
3795        bitmap4 supp_attrs;

3797        ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
3798        rp = VTOR4(vp);
3799        nfs4_init_stateid_types(&sid_types);

3801        /*
3802         * Only need to flush pages if there are any pages and
3803         * if the file is marked as dirty in some fashion.  The
3804         * file must be flushed so that we can accurately
3805         * determine the size of the file and the cached data
3806         * after the SETATTR returns.  A file is considered to
3807         * be dirty if it is either marked with R4DIRTY, has
3808         * outstanding i/o's active, or is mmap'd.  In this
3809         * last case, we can't tell whether there are dirty
3810         * pages, so we flush just to be sure.
3811         */
3812        if (nfs4_has_pages(vp) &&
3813            ((rp->r_flags & R4DIRTY) ||
3814            rp->r_count > 0 ||
3815            rp->r_mapcnt > 0)) {
3816                ASSERT(vp->v_type != VCHR);
3817                e.error = nfs4_putpage(vp, (offset_t)0, 0, 0, cr, NULL);
3818                if (e.error && (e.error == ENOSPC || e.error == EDQUOT)) {
3819                        mutex_enter(&rp->r_statelock);
3820                        if (!rp->r_error)
```

```
3821                                rp->r_error = e.error;
3822                        mutex_exit(&rp->r_statelock);
3823                }
3824        }

3826        if (mask & AT_SIZE) {
3827                /*
3828                 * Verification setattr compound for non-deleg AT_SIZE:
3829                 *      { Putfh; Getattr; Verify; Setattr; Getattr }
3830                 * Set ctime local here (outside the do_again label)
3831                 * so that subsequent retries (after failed VERIFY)
3832                 * will use ctime from GETATTR results (from failed
3833                 * verify compound) as VERIFY arg.
3834                 * If file has delegation, then VERIFY(time_metadata)
3835                 * is of little added value, so don't bother.
3836                 */
3837                mutex_enter(&rp->r_statev4_lock);
3838                if (rp->r_deleg_type == OPEN_DELEGATE_NONE ||
3839                    rp->r_deleg_return_pending) {
3840                        numops = 5;
3841                        ctime = rp->r_attr.va_ctime;
3842                }
3843                mutex_exit(&rp->r_statev4_lock);
3844        }

3846        recov_state.rs_flags = 0;
3847        recov_state.rs_num_retry_despite_err = 0;

3849        args.ctag = TAG_SETATTR;
3850 do_again:
3851 recov_retry:
3852        setattr_argop = numops - 2;

3854        args.array = argop;
3855        args.array_len = numops;

3857        e.error = nfs4_start_op(VTOMI4(vp), vp, NULL, &recov_state);
3858        if (e.error)
3859                return (e.error);


3862        /* putfh target fh */
3863        argop[0].argop = OP_CPUTFH;
3864        argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

3866        if (numops == 5) {
3867                /*
3868                 * We only care about the ctime, but need to get mtime
3869                 * and size for proper cache update.
3870                 */
3871                /* getattr */
3872                argop[1].argop = OP_GETATTR;
3873                argop[1].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
3874                argop[1].nfs_argop4_u.opgetattr.mi = VTOMI4(vp);

3876                /* verify - set later in loop */
3877                verify_argop = 2;
3878        }

3880        /* setattr */
3881        svp = rp->r_server;
3882        (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
3883        supp_attrs = svp->sv_supp_attrs;
3884        nfs_rw_exit(&svp->sv_lock);

3886        nfs4args_setattr(&argop[setattr_argop], vap, vsap, flags, rp, cr,
```



```
3887            supp_attrs, &e.error, &sid_types);
3888        stateid = argop[setattr_argop].nfs_argop4_u.opsetattr.stateid;
3889        if (e.error) {
3890                /* req time field(s) overflow - return immediately */
3891                nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state, needrecov);
3892                nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
3893                    opsetattr.obj_attributes);
3894                return (e.error);
3895        }
3896        omode = rp->r_attr.va_mode;

3898        /* getattr */
3899        argop[numops-1].argop = OP_GETATTR;
3900        argop[numops-1].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
3901        /*
3902         * If we are setting the ACL (indicated only by vsap != NULL), request
3903         * the ACL in this getattr.  The ACL returned from this getattr will be
3904         * used in updating the ACL cache.
3905         */
3906        if (vsap != NULL)
3907                argop[numops-1].nfs_argop4_u.opgetattr.attr_request |=
3908                    FATTR4_ACL_MASK;
3909        argop[numops-1].nfs_argop4_u.opgetattr.mi = VTOMI4(vp);

3911        /*
3912         * setattr iterates if the object size is set and the cached ctime
3913         * does not match the file ctime. In that case, verify the ctime first.
3914         */

3916        do {
3917                if (verify_argop != -1) {
3918                        /*
3919                         * Verify that the ctime match before doing setattr.
3920                         */
3921                        va.va_mask = AT_CTIME;
3922                        va.va_ctime = ctime;
3923                        svp = rp->r_server;
3924                        (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
3925                        supp_attrs = svp->sv_supp_attrs;
3926                        nfs_rw_exit(&svp->sv_lock);
3927                        e.error = nfs4args_verify(&argop[verify_argop], &va,
3928                            OP_VERIFY, supp_attrs);
3929                        if (e.error) {
3930                                /* req time field(s) overflow - return */
3931                                nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
3932                                    needrecov);
3933                                break;
3934                        }
3935                }

3937                doqueue = 1;

3939                t = gethrtime();

3941                rfs4call(VTOMI4(vp), &args, &res, cr, &doqueue, 0, &e);

3943                /*
3944                 * Purge the access cache and ACL cache if changing either the
3945                 * owner of the file, the group owner, or the mode.  These may
3946                 * change the access permissions of the file, so purge old
3947                 * information and start over again.
3948                 */
3949                if (mask & (AT_UID | AT_GID | AT_MODE)) {
3950                        (void) nfs4_access_purge_rp(rp);
3951                        if (rp->r_secattr != NULL) {
3952                                mutex_enter(&rp->r_statelock);
```

```
3953                                    vsp = rp->r_secattr;
3954                                    rp->r_secattr = NULL;
3955                                    mutex_exit(&rp->r_statelock);
3956                                    if (vsp != NULL)
3957                                            nfs4_acl_free_cache(vsp);
3958                            }
3959                    }

3961                    /*
3962                     * If res.array_len == numops, then everything succeeded,
3963                     * except for possibly the final getattr.  If only the
3964                     * last getattr failed, give up, and don't try recovery.
3965                     */
3966                    if (res.array_len == numops) {
3967                            nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
3968                                needrecov);
3969                            if (! e.error)
3970                                    resp = &res;
3971                            break;
3972                    }

3974                    /*
3975                     * if either rpc call failed or completely succeeded - done
3976                     */
3977                    needrecov = nfs4_needs_recovery(&e, FALSE, vp->v_vfsp);
3978                    if (e.error) {
3979                            PURGE_ATTRCACHE4(vp);
3980                            if (!needrecov) {
3981                                    nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
3982                                        needrecov);
3983                                    break;
3984                            }
3985                    }

3987                    /*
3988                     * Do proper retry for OLD_STATEID outside of the normal
3989                     * recovery framework.
3990                     */
3991                    if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
3992                        sid_types.cur_sid_type != SPEC_SID &&
3993                        sid_types.cur_sid_type != NO_SID) {
3994                            nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
3995                                needrecov);
3996                            nfs4_save_stateid(&stateid, &sid_types);
3997                            nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
3998                                opsetattr.obj_attributes);
3999                            if (verify_argop != -1) {
4000                                    nfs4args_verify_free(&argop[verify_argop]);
4001                                    verify_argop = -1;
4002                            }
4003                            (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4004                            goto recov_retry;
4005                    }

4007                    if (needrecov) {
4008                            bool_t abort;

4010                            abort = nfs4_start_recovery(&e,
4011                                VTOMI4(vp), vp, NULL, NULL, NULL,
4012                                OP_SETATTR, NULL, NULL, NULL);
4013                            nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
4014                                needrecov);
4015                            /*
4016                             * Do not retry if we failed with OLD_STATEID using
4017                             * a special stateid.  This is done to avoid looping
4018                             * with a broken server.
```



```
4019                             */
4020                            if (e.error == 0 && res.status == NFS4ERR_OLD_STATEID &&
4021                                (sid_types.cur_sid_type == SPEC_SID ||
4022                                sid_types.cur_sid_type == NO_SID))
4023                                    abort = TRUE;
4024                            if (!e.error) {
4025                                    if (res.status == NFS4ERR_BADOWNER)
4026                                            nfs4_log_badowner(VTOMI4(vp),
4027                                                OP_SETATTR);

4029                                    e.error = geterrno4(res.status);
4030                                    (void) xdr_free(xdr_COMPOUND4res_clnt,
4031                                        (caddr_t)&res);
4032                            }
4033                            nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4034                                opsetattr.obj_attributes);
4035                            if (verify_argop != -1) {
4036                                    nfs4args_verify_free(&argop[verify_argop]);
4037                                    verify_argop = -1;
4038                            }
4039                            if (abort == FALSE) {
4040                                    /*
4041                                     * Need to retry all possible stateids in
4042                                     * case the recovery error wasn't stateid
4043                                     * related or the stateids have become
4044                                     * stale (server reboot).
4045                                     */
4046                                    nfs4_init_stateid_types(&sid_types);
4047                                    goto recov_retry;
4048                            }
4049                            return (e.error);
4050                    }

4052                    /*
4053                     * Need to call nfs4_end_op before nfs4getattr to
4054                     * avoid potential nfs4_start_op deadlock. See RFE
4055                     * 4777612.  Calls to nfs4_invalidate_pages() and
4056                     * nfs4_purge_stale_fh() might also generate over the
4057                     * wire calls which my cause nfs4_start_op() deadlock.
4058                     */
4059                    nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state, needrecov);

4061                    /*
4062                     * Check to update lease.
4063                     */
4064                    resp = &res;
4065                    if (res.status == NFS4_OK) {
4066                            break;
4067                    }

4069                    /*
4070                     * Check if verify failed to see if try again
4071                     */
4072                    if ((verify_argop == -1) || (res.array_len != 3)) {
4073                            /*
4074                             * can't continue...
4075                             */
4076                            if (res.status == NFS4ERR_BADOWNER)
4077                                    nfs4_log_badowner(VTOMI4(vp), OP_SETATTR);

4079                            e.error = geterrno4(res.status);
4080                    } else {
4081                            /*
4082                             * When the verify request fails, the client ctime is
4083                             * not in sync with the server. This is the same as
4084                             * the version 3 "not synchronized" error, and we
```

```
4085                         * handle it in a similar manner (XXX do we need to???).
4086                         * Use the ctime returned in the first getattr for
4087                         * the input to the next verify.
4088                         * If we couldn't get the attributes, then we give up
4089                         * because we can't complete the operation as required.
4090                         */
4091                        garp = &res.array[1].nfs_resop4_u.opgetattr.ga_res;
4092                }
4093                if (e.error) {
4094                        PURGE_ATTRCACHE4(vp);
4095                        nfs4_purge_stale_fh(e.error, vp, cr);
4096                } else {
4097                        /*
4098                         * retry with a new verify value
4099                         */
4100                        ctime = garp->n4g_va.va_ctime;
4101                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4102                        resp = NULL;
4103                }
4104                if (!e.error) {
4105                        nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4106                            opsetattr.obj_attributes);
4107                        if (verify_argop != -1) {
4108                                nfs4args_verify_free(&argop[verify_argop]);
4109                                verify_argop = -1;
4110                        }
4111                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4112                        goto do_again;
4113                }
4114        } while (!e.error);

4116        if (e.error) {
4117                /*
4118                 * If we are here, rfs4call has an irrecoverable error - return
4119                 */
4120                nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4121                    opsetattr.obj_attributes);
4122                if (verify_argop != -1) {
4123                        nfs4args_verify_free(&argop[verify_argop]);
4124                        verify_argop = -1;
4125                }
4126                if (resp)
4127                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
4128                return (e.error);
4129        }



4133        /*
4134         * If changing the size of the file, invalidate
4135         * any local cached data which is no longer part
4136         * of the file.  We also possibly invalidate the
4137         * last page in the file.  We could use
4138         * pvn_vpzero(), but this would mark the page as
4139         * modified and require it to be written back to
4140         * the server for no particularly good reason.
4141         * This way, if we access it, then we bring it
4142         * back in.  A read should be cheaper than a
4143         * write.
4144         */
4145        if (mask & AT_SIZE) {
4146                nfs4_invalidate_pages(vp, (vap->va_size & PAGEMASK), cr);
4147        }

4149        /* either no error or one of the postop getattr failed */
```



```
4151        /*
4152         * XXX Perform a simplified version of wcc checking. Instead of
4153         * have another getattr to get pre-op, just purge cache if
4154         * any of the ops prior to and including the getattr failed.
4155         * If the getattr succeeded then update the attrcache accordingly.
4156         */

4158        garp = NULL;
4159        if (res.status == NFS4_OK) {
4160                /*
4161                 * Last getattr
4162                 */
4163                resop = &res.array[numops - 1];
4164                garp = &resop->nfs_resop4_u.opgetattr.ga_res;
4165        }
4166        /*
4167         * In certain cases, nfs4_update_attrcache() will purge the attrcache,
4168         * rather than filling it.  See the function itself for details.
4169         */
4170        e.error = nfs4_update_attrcache(res.status, garp, t, vp, cr);
4171        if (garp != NULL) {
4172                if (garp->n4g_resbmap & FATTR4_ACL_MASK) {
4173                        nfs4_acl_fill_cache(rp, &garp->n4g_vsa);
4174                        vs_ace4_destroy(&garp->n4g_vsa);
4175                } else {
4176                        if (vsap != NULL) {
4177                                /*
4178                                 * The ACL was supposed to be set and to be
4179                                 * returned in the last getattr of this
4180                                 * compound, but for some reason the getattr
4181                                 * result doesn't contain the ACL.  In this
4182                                 * case, purge the ACL cache.
4183                                 */
4184                                if (rp->r_secattr != NULL) {
4185                                        mutex_enter(&rp->r_statelock);
4186                                        vsp = rp->r_secattr;
4187                                        rp->r_secattr = NULL;
4188                                        mutex_exit(&rp->r_statelock);
4189                                        if (vsp != NULL)
4190                                                nfs4_acl_free_cache(vsp);
4191                                }
4192                        }
4193                }
4194        }

4196        if (res.status == NFS4_OK && (mask & AT_SIZE)) {
4197                /*
4198                 * Set the size, rather than relying on getting it updated
4199                 * via a GETATTR.  With delegations the client tries to
4200                 * suppress GETATTR calls.
4201                 */
4202                mutex_enter(&rp->r_statelock);
4203                rp->r_size = vap->va_size;
4204                mutex_exit(&rp->r_statelock);
4205        }

4207        /*
4208         * Can free up request args and res
4209         */
4210        nfs4_fattr4_free(&argop[setattr_argop].nfs_argop4_u.
4211            opsetattr.obj_attributes);
4212        if (verify_argop != -1) {
4213                nfs4args_verify_free(&argop[verify_argop]);
4214                verify_argop = -1;
4215        }
4216        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
```

```
4218         /*
4219          * Some servers will change the mode to clear the setuid
4220          * and setgid bits when changing the uid or gid.  The
4221          * client needs to compensate appropriately.
4222          */
4223         if (mask & (AT_UID | AT_GID)) {
4224                 int terror, do_setattr;

4226                 do_setattr = 0;
4227                 va.va_mask = AT_MODE;
4228                 terror = nfs4getattr(vp, &va, cr);
4229                 if (!terror &&
4230                     (((mask & AT_MODE) && va.va_mode != vap->va_mode) ||
4231                     (!(mask & AT_MODE) && va.va_mode != omode))) {
4232                         va.va_mask = AT_MODE;
4233                         if (mask & AT_MODE) {
4234                                 /*
4235                                  * We asked the mode to be changed and what
4236                                  * we just got from the server in getattr is
4237                                  * not what we wanted it to be, so set it now.
4238                                  */
4239                                 va.va_mode = vap->va_mode;
4240                                 do_setattr = 1;
4241                         } else {
4242                                 /*
4243                                  * We did not ask the mode to be changed,
4244                                  * Check to see that the server just cleared
4245                                  * I_SUID and I_GUID from it. If not then
4246                                  * set mode to omode with UID/GID cleared.
4247                                  */
4248                                 if (nfs4_compare_modes(va.va_mode, omode)) {
4249                                         omode &= ~(S_ISUID|S_ISGID);
4250                                         va.va_mode = omode;
4251                                         do_setattr = 1;
4252                                 }
4253                         }

4255                         if (do_setattr)
4256                                 (void) nfs4setattr(vp, &va, 0, cr, NULL);
4257                 }
4258         }

4260         return (e.error);
4261 }

4263 /* ARGSUSED */
4264 static int
4265 nfs4_access(vnode_t *vp, int mode, int flags, cred_t *cr, caller_context_t *ct)
4266 {
4267         COMPOUND4args_clnt args;
4268         COMPOUND4res_clnt res;
4269         int doqueue;
4270         uint32_t acc, resacc, argacc;
4271         rnode4_t *rp;
4272         cred_t *cred, *ncr, *ncrfree = NULL;
4273         nfs4_access_type_t cacc;
4274         int num_ops;
4275         nfs_argop4 argop[3];
4276         nfs_resop4 *resop;
4277         bool_t needrecov = FALSE, do_getattr;
4278         nfs4_recov_state_t recov_state;
4279         int rpc_error;
4280         hrtime_t t;
4281         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
4282         mntinfo4_t *mi = VTOMI4(vp);
```



```
4284         if (nfs_zone() != mi->mi_zone)
4285                 return (EIO);

4287         acc = 0;
4288         if (mode & VREAD)
4289                 acc |= ACCESS4_READ;
4290         if (mode & VWRITE) {
4291                 if ((vp->v_vfsp->vfs_flag & VFS_RDONLY) && !ISVDEV(vp->v_type))
4292                         return (EROFS);
4293                 if (vp->v_type == VDIR)
4294                         acc |= ACCESS4_DELETE;
4295                 acc |= ACCESS4_MODIFY | ACCESS4_EXTEND;
4296         }
4297         if (mode & VEXEC) {
4298                 if (vp->v_type == VDIR)
4299                         acc |= ACCESS4_LOOKUP;
4300                 else
4301                         acc |= ACCESS4_EXECUTE;
4302         }

4304         if (VTOR4(vp)->r_acache != NULL) {
4305                 e.error = nfs4_validate_caches(vp, cr);
4306                 if (e.error)
4307                         return (e.error);
4308         }

4310         rp = VTOR4(vp);
4311         if (vp->v_type == VDIR)
4312                 argacc = ACCESS4_READ | ACCESS4_DELETE | ACCESS4_MODIFY |
4313                     ACCESS4_EXTEND | ACCESS4_LOOKUP;
4314         else
4315                 argacc = ACCESS4_READ | ACCESS4_MODIFY | ACCESS4_EXTEND |
4316                     ACCESS4_EXECUTE;
4317         recov_state.rs_flags = 0;
4318         recov_state.rs_num_retry_despite_err = 0;

4320         cred = cr;
4321         /*
4322          * ncr and ncrfree both initially
4323          * point to the memory area returned
4324          * by crnetadjust();
4325          * ncrfree not NULL when exiting means
4326          * that we need to release it
4327          */
4328         ncr = crnetadjust(cred);
4329         ncrfree = ncr;

4331 tryagain:
4332         cacc = nfs4_access_check(rp, acc, cred);
4333         if (cacc == NFS4_ACCESS_ALLOWED) {
4334                 if (ncrfree != NULL)
4335                         crfree(ncrfree);
4336                 return (0);
4337         }
4338         if (cacc == NFS4_ACCESS_DENIED) {
4339                 /*
4340                  * If the cred can be adjusted, try again
4341                  * with the new cred.
4342                  */
4343                 if (ncr != NULL) {
4344                         cred = ncr;
4345                         ncr = NULL;
4346                         goto tryagain;
4347                 }
4348                 if (ncrfree != NULL)
```

```
4349                        crfree(ncrfree);
4350                return (EACCES);
4351        }

4353 recov_retry:
4354        /*
4355         * Don't take with r_statev4_lock here. r_deleg_type could
4356         * change as soon as lock is released.  Since it is an int,
4357         * there is no atomicity issue.
4358         */
4359        do_getattr = (rp->r_deleg_type == OPEN_DELEGATE_NONE);
4360        num_ops = do_getattr ? 3 : 2;

4362        args.ctag = TAG_ACCESS;

4364        args.array_len = num_ops;
4365        args.array = argop;

4367        if (e.error = nfs4_start_fop(mi, vp, NULL, OH_ACCESS,
4368            &recov_state, NULL)) {
4369                if (ncrfree != NULL)
4370                        crfree(ncrfree);
4371                return (e.error);
4372        }

4374        /* putfh target fh */
4375        argop[0].argop = OP_CPUTFH;
4376        argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(vp)->r_fh;

4378        /* access */
4379        argop[1].argop = OP_ACCESS;
4380        argop[1].nfs_argop4_u.opaccess.access = argacc;

4382        /* getattr */
4383        if (do_getattr) {
4384                argop[2].argop = OP_GETATTR;
4385                argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
4386                argop[2].nfs_argop4_u.opgetattr.mi = mi;
4387        }

4389        NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
4390            "nfs4_access: %s call, rp %s", needrecov ? "recov" : "first",
4391            rnode4info(VTOR4(vp))));

4393        doqueue = 1;
4394        t = gethrtime();
4395        rfs4call(VTOMI4(vp), &args, &res, cred, &doqueue, 0, &e);
4396        rpc_error = e.error;

4398        needrecov = nfs4_needs_recovery(&e, FALSE, vp->v_vfsp);
4399        if (needrecov) {
4400                NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
4401                    "nfs4_access: initiating recovery\n"));

4403                if (nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
4404                    NULL, OP_ACCESS, NULL, NULL, NULL) == FALSE) {
4405                        nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_ACCESS,
4406                            &recov_state, needrecov);
4407                        if (!e.error)
4408                                (void) xdr_free(xdr_COMPOUND4res_clnt,
4409                                    (caddr_t)&res);
4410                        goto recov_retry;
4411                }
4412        }
4413        nfs4_end_fop(mi, vp, NULL, OH_ACCESS, &recov_state, needrecov);
```



```
4415        if (e.error)
4416                goto out;

4418        if (res.status) {
4419                e.error = geterrno4(res.status);
4420                /*
4421                 * This might generate over the wire calls throught
4422                 * nfs4_invalidate_pages. Hence we need to call nfs4_end_op()
4423                 * here to avoid a deadlock.
4424                 */
4425                nfs4_purge_stale_fh(e.error, vp, cr);
4426                goto out;
4427        }
4428        resop = &res.array[1];  /* access res */

4430        resacc = resop->nfs_resop4_u.opaccess.access;

4432        if (do_getattr) {
4433                resop++;        /* getattr res */
4434                nfs4_attr_cache(vp, &resop->nfs_resop4_u.opgetattr.ga_res,
4435                    t, cr, FALSE, NULL);
4436        }

4438        if (!e.error) {
4439                nfs4_access_cache(rp, argacc, resacc, cred);
4440                /*
4441                 * we just cached results with cred; if cred is the
4442                 * adjusted credentials from crnetadjust, we do not want
4443                 * to release them before exiting: hence setting ncrfree
4444                 * to NULL
4445                 */
4446                if (cred != cr)
4447                        ncrfree = NULL;
4448                /* XXX check the supported bits too? */
4449                if ((acc & resacc) != acc) {
4450                        /*
4451                         * The following code implements the semantic
4452                         * that a setuid root program has *at least* the
4453                         * permissions of the user that is running the
4454                         * program.  See rfs3call() for more portions
4455                         * of the implementation of this functionality.
4456                         */
4457                        /* XXX-LP */
4458                        if (ncr != NULL) {
4459                                (void) xdr_free(xdr_COMPOUND4res_clnt,
4460                                    (caddr_t)&res);
4461                                cred = ncr;
4462                                ncr = NULL;
4463                                goto tryagain;
4464                        }
4465                        e.error = EACCES;
4466                }
4467        }

4469 out:
4470        if (!rpc_error)
4471                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

4473        if (ncrfree != NULL)
4474                crfree(ncrfree);

4476        return (e.error);
4477 }

4479 /* ARGSUSED */
4480 static int
```

```
4481 nfs4_readlink(vnode_t *vp, struct uio *uiop, cred_t *cr, caller_context_t *ct)
4482 {
4483         COMPOUND4args_clnt args;
4484         COMPOUND4res_clnt res;
4485         int doqueue;
4486         rnode4_t *rp;
4487         nfs_argop4 argop[3];
4488         nfs_resop4 *resop;
4489         READLINK4res *lr_res;
4490         nfs4_ga_res_t *garp;
4491         uint_t len;
4492         char *linkdata;
4493         bool_t needrecov = FALSE;
4494         nfs4_recov_state_t recov_state;
4495         hrtime_t t;
4496         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

4498         if (nfs_zone() != VTOMI4(vp)->mi_zone)
4499                 return (EIO);
4500         /*
4501          * Can't readlink anything other than a symbolic link.
4502          */
4503         if (vp->v_type != VLNK)
4504                 return (EINVAL);

4506         rp = VTOR4(vp);
4507         if (nfs4_do_symlink_cache && rp->r_symlink.contents != NULL) {
4508                 e.error = nfs4_validate_caches(vp, cr);
4509                 if (e.error)
4510                         return (e.error);
4511                 mutex_enter(&rp->r_statelock);
4512                 if (rp->r_symlink.contents != NULL) {
4513                         e.error = uiomove(rp->r_symlink.contents,
4514                             rp->r_symlink.len, UIO_READ, uiop);
4515                         mutex_exit(&rp->r_statelock);
4516                         return (e.error);
4517                 }
4518                 mutex_exit(&rp->r_statelock);
4519         }
4520         recov_state.rs_flags = 0;
4521         recov_state.rs_num_retry_despite_err = 0;

4523 recov_retry:
4524         args.array_len = 3;
4525         args.array = argop;
4526         args.ctag = TAG_READLINK;

4528         e.error = nfs4_start_op(VTOMI4(vp), vp, NULL, &recov_state);
4529         if (e.error) {
4530                 return (e.error);
4531         }

4533         /* 0. putfh symlink fh */
4534         argop[0].argop = OP_CPUTFH;
4535         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(vp)->r_fh;

4537         /* 1. readlink */
4538         argop[1].argop = OP_READLINK;

4540         /* 2. getattr */
4541         argop[2].argop = OP_GETATTR;
4542         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
4543         argop[2].nfs_argop4_u.opgetattr.mi = VTOMI4(vp);

4545         doqueue = 1;
```



```
4547         NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
4548             "nfs4_readlink: %s call, rp %s", needrecov ? "recov" : "first",
4549             rnode4info(VTOR4(vp))));

4551         t = gethrtime();

4553         rfs4call(VTOMI4(vp), &args, &res, cr, &doqueue, 0, &e);

4555         needrecov = nfs4_needs_recovery(&e, FALSE, vp->v_vfsp);
4556         if (needrecov) {
4557                 NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
4558                     "nfs4_readlink: initiating recovery\n"));

4560                 if (nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
4561                     NULL, OP_READLINK, NULL, NULL, NULL) == FALSE) {
4562                         if (!e.error)
4563                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
4564                                     (caddr_t)&res);

4566                         nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state,
4567                             needrecov);
4568                         goto recov_retry;
4569                 }
4570         }

4572         nfs4_end_op(VTOMI4(vp), vp, NULL, &recov_state, needrecov);

4574         if (e.error)
4575                 return (e.error);

4577         /*
4578          * There is an path in the code below which calls
4579          * nfs4_purge_stale_fh(), which may generate otw calls through
4580          * nfs4_invalidate_pages. Hence we need to call nfs4_end_op()
4581          * here to avoid nfs4_start_op() deadlock.
4582          */

4584         if (res.status && (res.array_len < args.array_len)) {
4585                 /*
4586                  * either Putfh or Link failed
4587                  */
4588                 e.error = geterrno4(res.status);
4589                 nfs4_purge_stale_fh(e.error, vp, cr);
4590                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
4591                 return (e.error);
4592         }

4594         resop = &res.array[1];  /* readlink res */
4595         lr_res = &resop->nfs_resop4_u.opreadlink;

4597         /*
4598          * treat symlink names as data
4599          */
4600         linkdata = utf8_to_str(&lr_res->link, &len, NULL);
4601         if (linkdata != NULL) {
4602                 int uio_len = len - 1;
4603                 /* len includes null byte, which we won't uiomove */
4604                 e.error = uiomove(linkdata, uio_len, UIO_READ, uiop);
4605                 if (nfs4_do_symlink_cache && rp->r_symlink.contents == NULL) {
4606                         mutex_enter(&rp->r_statelock);
4607                         if (rp->r_symlink.contents == NULL) {
4608                                 rp->r_symlink.contents = linkdata;
4609                                 rp->r_symlink.len = uio_len;
4610                                 rp->r_symlink.size = len;
4611                                 mutex_exit(&rp->r_statelock);
4612                         } else {
```

```
4613                                mutex_exit(&rp->r_statelock);
4614                                kmem_free(linkdata, len);
4615                        }
4616                } else {
4617                        kmem_free(linkdata, len);
4618                }
4619        }
4620        if (res.status == NFS4_OK) {
4621                resop++;        /* getattr res */
4622                garp = &resop->nfs_resop4_u.opgetattr.ga_res;
4623        }
4624        e.error = nfs4_update_attrcache(res.status, garp, t, vp, cr);

4626        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

4628        /*
4629         * The over the wire error for attempting to readlink something
4630         * other than a symbolic link is ENXIO.  However, we need to
4631         * return EINVAL instead of ENXIO, so we map it here.
4632         */
4633        return (e.error == ENXIO ? EINVAL : e.error);
4634 }

4636 /*
4637  * Flush local dirty pages to stable storage on the server.
4638  *
4639  * If FNODSYNC is specified, then there is nothing to do because
4640  * metadata changes are not cached on the client before being
4641  * sent to the server.
4642  */
4643 /* ARGSUSED */
4644 static int
4645 nfs4_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
4646 {
4647        int error;

4649        if ((syncflag & FNODSYNC) || IS_SWAPVP(vp))
4650                return (0);
4651        if (nfs_zone() != VTOMI4(vp)->mi_zone)
4652                return (EIO);
4653        error = nfs4_putpage_commit(vp, (offset_t)0, 0, cr);
4654        if (!error)
4655                error = VTOR4(vp)->r_error;
4656        return (error);
4657 }

4659 /*
4660  * Weirdness: if the file was removed or the target of a rename
4661  * operation while it was open, it got renamed instead.  Here we
4662  * remove the renamed file.
4663  */
4664 /* ARGSUSED */
4665 void
4666 nfs4_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
4667 {
4668        rnode4_t *rp;

4670        ASSERT(vp != DNLC_NO_VNODE);

4672        rp = VTOR4(vp);

4674        if (IS_SHADOW(vp, rp)) {
4675                sv_inactive(vp);
4676                return;
4677        }
```



```
4679        /*
4680         * If this is coming from the wrong zone, we let someone in the right
4681         * zone take care of it asynchronously.  We can get here due to
4682         * VN_RELE() being called from pageout() or fsflush().  This call may
4683         * potentially turn into an expensive no-op if, for instance, v_count
4684         * gets incremented in the meantime, but it's still correct.
4685         */
4686        if (nfs_zone() != VTOMI4(vp)->mi_zone) {
4687                nfs4_async_inactive(vp, cr);
4688                return;
4689        }

4691        /*
4692         * Some of the cleanup steps might require over-the-wire
4693         * operations.  Since VOP_INACTIVE can get called as a result of
4694         * other over-the-wire operations (e.g., an attribute cache update
4695         * can lead to a DNLC purge), doing those steps now would lead to a
4696         * nested call to the recovery framework, which can deadlock.  So
4697         * do any over-the-wire cleanups asynchronously, in a separate
4698         * thread.
4699         */

4701        mutex_enter(&rp->r_os_lock);
4702        mutex_enter(&rp->r_statelock);
4703        mutex_enter(&rp->r_statev4_lock);

4705        if (vp->v_type == VREG && list_head(&rp->r_open_streams) != NULL) {
4706                mutex_exit(&rp->r_statev4_lock);
4707                mutex_exit(&rp->r_statelock);
4708                mutex_exit(&rp->r_os_lock);
4709                nfs4_async_inactive(vp, cr);
4710                return;
4711        }

4713        if (rp->r_deleg_type == OPEN_DELEGATE_READ ||
4714            rp->r_deleg_type == OPEN_DELEGATE_WRITE) {
4715                mutex_exit(&rp->r_statev4_lock);
4716                mutex_exit(&rp->r_statelock);
4717                mutex_exit(&rp->r_os_lock);
4718                nfs4_async_inactive(vp, cr);
4719                return;
4720        }

4722        if (rp->r_unldvp != NULL) {
4723                mutex_exit(&rp->r_statev4_lock);
4724                mutex_exit(&rp->r_statelock);
4725                mutex_exit(&rp->r_os_lock);
4726                nfs4_async_inactive(vp, cr);
4727                return;
4728        }
4729        mutex_exit(&rp->r_statev4_lock);
4730        mutex_exit(&rp->r_statelock);
4731        mutex_exit(&rp->r_os_lock);

4733        rp4_addfree(rp, cr);
4734 }

4736 /*
4737  * nfs4_inactive_otw - nfs4_inactive, plus over-the-wire calls to free up
4738  * various bits of state.  The caller must not refer to vp after this call.
4739  */

4741 void
4742 nfs4_inactive_otw(vnode_t *vp, cred_t *cr)
4743 {
4744        rnode4_t *rp = VTOR4(vp);
```

```
4745         nfs4_recov_state_t recov_state;
4746         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
4747         vnode_t *unldvp;
4748         char *unlname;
4749         cred_t *unlcred;
4750         COMPOUND4args_clnt args;
4751         COMPOUND4res_clnt res, *resp;
4752         nfs_argop4 argop[2];
4753         int doqueue;
4754 #ifdef DEBUG
4755         char *name;
4756 #endif

4758         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
4759         ASSERT(!IS_SHADOW(vp, rp));

4761 #ifdef DEBUG
4762         name = fn_name(VTOSV(vp)->sv_name);
4763         NFS4_DEBUG(nfs4_client_inactive_debug, (CE_NOTE, "nfs4_inactive_otw: "
4764             "release vnode %s", name));
4765         kmem_free(name, MAXNAMELEN);
4766 #endif

4768         if (vp->v_type == VREG) {
4769                 bool_t recov_failed = FALSE;

4771                 e.error = nfs4close_all(vp, cr);
4772                 if (e.error) {
4773                         /* Check to see if recovery failed */
4774                         mutex_enter(&(VTOMI4(vp)->mi_lock));
4775                         if (VTOMI4(vp)->mi_flags & MI4_RECOV_FAIL)
4776                                 recov_failed = TRUE;
4777                         mutex_exit(&(VTOMI4(vp)->mi_lock));
4778                         if (!recov_failed) {
4779                                 mutex_enter(&rp->r_statelock);
4780                                 if (rp->r_flags & R4RECOVERR)
4781                                         recov_failed = TRUE;
4782                                 mutex_exit(&rp->r_statelock);
4783                         }
4784                         if (recov_failed) {
4785                                 NFS4_DEBUG(nfs4_client_recov_debug,
4786                                     (CE_NOTE, "nfs4_inactive_otw: "
4787                                     "close failed (recovery failure)"));
4788                         }
4789                 }
4790         }

4792 redo:
4793         if (rp->r_unldvp == NULL) {
4794                 rp4_addfree(rp, cr);
4795                 return;
4796         }

4798         /*
4799          * Save the vnode pointer for the directory where the
4800          * unlinked-open file got renamed, then set it to NULL
4801          * to prevent another thread from getting here before
4802          * we're done with the remove.  While we have the
4803          * statelock, make local copies of the pertinent rnode
4804          * fields.  If we weren't to do this in an atomic way, the
4805          * the unl* fields could become inconsistent with respect
4806          * to each other due to a race condition between this
4807          * code and nfs_remove().  See bug report 1034328.
4808          */
4809         mutex_enter(&rp->r_statelock);
4810         if (rp->r_unldvp == NULL) {
```



```
4811                 mutex_exit(&rp->r_statelock);
4812                 rp4_addfree(rp, cr);
4813                 return;
4814         }

4816         unldvp = rp->r_unldvp;
4817         rp->r_unldvp = NULL;
4818         unlname = rp->r_unlname;
4819         rp->r_unlname = NULL;
4820         unlcred = rp->r_unlcred;
4821         rp->r_unlcred = NULL;
4822         mutex_exit(&rp->r_statelock);

4824         /*
4825          * If there are any dirty pages left, then flush
4826          * them.  This is unfortunate because they just
4827          * may get thrown away during the remove operation,
4828          * but we have to do this for correctness.
4829          */
4830         if (nfs4_has_pages(vp) &&
4831             ((rp->r_flags & R4DIRTY) || rp->r_count > 0)) {
4832                 ASSERT(vp->v_type != VCHR);
4833                 e.error = nfs4_putpage(vp, (u_offset_t)0, 0, 0, cr, NULL);
4834                 if (e.error) {
4835                         mutex_enter(&rp->r_statelock);
4836                         if (!rp->r_error)
4837                                 rp->r_error = e.error;
4838                         mutex_exit(&rp->r_statelock);
4839                 }
4840         }

4842         recov_state.rs_flags = 0;
4843         recov_state.rs_num_retry_despite_err = 0;
4844 recov_retry_remove:
4845         /*
4846          * Do the remove operation on the renamed file
4847          */
4848         args.ctag = TAG_INACTIVE;

4850         /*
4851          * Remove ops: putfh dir; remove
4852          */
4853         args.array_len = 2;
4854         args.array = argop;

4856         e.error = nfs4_start_op(VTOMI4(unldvp), unldvp, NULL, &recov_state);
4857         if (e.error) {
4858                 kmem_free(unlname, MAXNAMELEN);
4859                 crfree(unlcred);
4860                 VN_RELE(unldvp);
4861                 /*
4862                  * Try again; this time around r_unldvp will be NULL, so we'll
4863                  * just call rp4_addfree() and return.
4864                  */
4865                 goto redo;
4866         }

4868         /* putfh directory */
4869         argop[0].argop = OP_CPUTFH;
4870         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(unldvp)->r_fh;

4872         /* remove */
4873         argop[1].argop = OP_CREMOVE;
4874         argop[1].nfs_argop4_u.opcremove.ctarget = unlname;

4876         doqueue = 1;
```

```
4877         resp = &res;

4879 #if 0 /* notyet */
4880         /*
4881          * Can't do this yet.  We may be being called from
4882          * dnlc_purge_XXX while that routine is holding a
4883          * mutex lock to the nc_rele list.  The calls to
4884          * nfs3_cache_wcc_data may result in calls to
4885          * dnlc_purge_XXX.  This will result in a deadlock.
4886          */
4887         rfs4call(VTOMI4(unldvp), &args, &res, unlcred, &doqueue, 0, &e);
4888         if (e.error) {
4889                 PURGE_ATTRCACHE4(unldvp);
4890                 resp = NULL;
4891         } else if (res.status) {
4892                 e.error = geterrno4(res.status);
4893                 PURGE_ATTRCACHE4(unldvp);
4894                 /*
4895                  * This code is inactive right now
4896                  * but if made active there should
4897                  * be a nfs4_end_op() call before
4898                  * nfs4_purge_stale_fh to avoid start_op()
4899                  * deadlock. See BugId: 4948726
4900                  */
4901                 nfs4_purge_stale_fh(error, unldvp, cr);
4902         } else {
4903                 nfs_resop4 *resop;
4904                 REMOVE4res *rm_res;

4906                 resop = &res.array[1];
4907                 rm_res = &resop->nfs_resop4_u.opremove;
4908                 /*
4909                  * Update directory cache attribute,
4910                  * readdir and dnlc caches.
4911                  */
4912                 nfs4_update_dircaches(&rm_res->cinfo, unldvp, NULL, NULL, NULL);
4913         }
4914 #else
4915         rfs4call(VTOMI4(unldvp), &args, &res, unlcred, &doqueue, 0, &e);

4917         PURGE_ATTRCACHE4(unldvp);
4918 #endif

4920         if (nfs4_needs_recovery(&e, FALSE, unldvp->v_vfsp)) {
4921                 if (nfs4_start_recovery(&e, VTOMI4(unldvp), unldvp, NULL,
4922                     NULL, NULL, OP_REMOVE, NULL, NULL, NULL) == FALSE) {
4923                         if (!e.error)
4924                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
4925                                     (caddr_t)&res);
4926                         nfs4_end_op(VTOMI4(unldvp), unldvp, NULL,
4927                             &recov_state, TRUE);
4928                         goto recov_retry_remove;
4929                 }
4930         }
4931         nfs4_end_op(VTOMI4(unldvp), unldvp, NULL, &recov_state, FALSE);

4933         /*
4934          * Release stuff held for the remove
4935          */
4936         VN_RELE(unldvp);
4937         if (!e.error && resp)
4938                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);

4940         kmem_free(unlname, MAXNAMELEN);
4941         crfree(unlcred);
4942         goto redo;
```



```
4943 }

4945 /*
4946  * Remote file system operations having to do with directory manipulation.
4947  */
4948 /* ARGSUSED3 */
4949 int
4950 nfs4_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
4951     int flags, vnode_t *rdir, cred_t *cr, caller_context_t *ct,
4952     int *direntflags, pathname_t *realpnp)
4953 {
4954         int error;
4955         vnode_t *vp, *avp = NULL;
4956         rnode4_t *drp;

4958         *vpp = NULL;
4959         if (nfs_zone() != VTOMI4(dvp)->mi_zone)
4960                 return (EPERM);
4961         /*
4962          * if LOOKUP_XATTR, must replace dvp (object) with
4963          * object's attrdir before continuing with lookup
4964          */
4965         if (flags & LOOKUP_XATTR) {
4966                 error = nfs4lookup_xattr(dvp, nm, &avp, flags, cr);
4967                 if (error)
4968                         return (error);

4970                 dvp = avp;

4972                 /*
4973                  * If lookup is for "", just return dvp now.  The attrdir
4974                  * has already been activated (from nfs4lookup_xattr), and
4975                  * the caller will RELE the original dvp -- not
4976                  * the attrdir.  So, set vpp and return.
4977                  * Currently, when the LOOKUP_XATTR flag is
4978                  * passed to VOP_LOOKUP, the name is always empty, and
4979                  * shortcircuiting here avoids 3 unneeded lock/unlock
4980                  * pairs.
4981                  *
4982                  * If a non-empty name was provided, then it is the
4983                  * attribute name, and it will be looked up below.
4984                  */
4985                 if (*nm == '\0') {
4986                         *vpp = dvp;
4987                         return (0);
4988                 }

4990                 /*
4991                  * The vfs layer never sends a name when asking for the
4992                  * attrdir, so we should never get here (unless of course
4993                  * name is passed at some time in future -- at which time
4994                  * we'll blow up here).
4995                  */
4996                 ASSERT(0);
4997         }

4999         drp = VTOR4(dvp);
5000         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR4(dvp)))
5001                 return (EINTR);

5003         error = nfs4lookup(dvp, nm, vpp, cr, 0);
5004         nfs_rw_exit(&drp->r_rwlock);

5006         /*
5007          * If vnode is a device, create special vnode.
5008          */
```

```
5009         if (!error && ISVDEV((*vpp)->v_type)) {
5010                 vp = *vpp;
5011                 *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
5012                 VN_RELE(vp);
5013         }

5015         return (error);
5016 }

5018 /* ARGSUSED */
5019 static int
5020 nfs4lookup_xattr(vnode_t *dvp, char *nm, vnode_t **vpp, int flags, cred_t *cr)
5021 {
5022         int error;
5023         rnode4_t *drp;
5024         int cflag = ((flags & CREATE_XATTR_DIR) != 0);
5025         mntinfo4_t *mi;

5027         mi = VTOMI4(dvp);
5028         if (!(mi->mi_vfsp->vfs_flag & VFS_XATTR) &&
5029             !vfs_has_feature(mi->mi_vfsp, VFSFT_SYSATTR_VIEWS))
5030                 return (EINVAL);

5032         drp = VTOR4(dvp);
5033         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR4(dvp)))
5034                 return (EINTR);

5036         mutex_enter(&drp->r_statelock);
5037         /*
5038          * If the server doesn't support xattrs just return EINVAL
5039          */
5040         if (drp->r_xattr_dir == NFS4_XATTR_DIR_NOTSUPP) {
5041                 mutex_exit(&drp->r_statelock);
5042                 nfs_rw_exit(&drp->r_rwlock);
5043                 return (EINVAL);
5044         }

5046         /*
5047          * If there is a cached xattr directory entry,
5048          * use it as long as the attributes are valid. If the
5049          * attributes are not valid, take the simple approach and
5050          * free the cached value and re-fetch a new value.
5051          *
5052          * We don't negative entry cache for now, if we did we
5053          * would need to check if the file has changed on every
5054          * lookup. But xattrs don't exist very often and failing
5055          * an openattr is not much more expensive than and NVERIFY or GETATTR
5056          * so do an openattr over the wire for now.
5057          */
5058         if (drp->r_xattr_dir != NULL) {
5059                 if (ATTRCACHE4_VALID(dvp)) {
5060                         VN_HOLD(drp->r_xattr_dir);
5061                         *vpp = drp->r_xattr_dir;
5062                         mutex_exit(&drp->r_statelock);
5063                         nfs_rw_exit(&drp->r_rwlock);
5064                         return (0);
5065                 }
5066                 VN_RELE(drp->r_xattr_dir);
5067                 drp->r_xattr_dir = NULL;
5068         }
5069         mutex_exit(&drp->r_statelock);

5071         error = nfs4openattr(dvp, vpp, cflag, cr);

5073         nfs_rw_exit(&drp->r_rwlock);
```



```
5075         return (error);
5076 }

5078 static int
5079 nfs4lookup(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr, int skipdnlc)
5080 {
5081         int error;
5082         rnode4_t *drp;

5084         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);

5086         /*
5087          * If lookup is for "", just return dvp.  Don't need
5088          * to send it over the wire, look it up in the dnlc,
5089          * or perform any access checks.
5090          */
5091         if (*nm == '\0') {
5092                 VN_HOLD(dvp);
5093                 *vpp = dvp;
5094                 return (0);
5095         }

5097         /*
5098          * Can't do lookups in non-directories.
5099          */
5100         if (dvp->v_type != VDIR)
5101                 return (ENOTDIR);

5103         /*
5104          * If lookup is for ".", just return dvp.  Don't need
5105          * to send it over the wire or look it up in the dnlc,
5106          * just need to check access.
5107          */
5108         if (nm[0] == '.' && nm[1] == '\0') {
5109                 error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5110                 if (error)
5111                         return (error);
5112                 VN_HOLD(dvp);
5113                 *vpp = dvp;
5114                 return (0);
5115         }

5117         drp = VTOR4(dvp);
5118         if (!(drp->r_flags & R4LOOKUP)) {
5119                 mutex_enter(&drp->r_statelock);
5120                 drp->r_flags |= R4LOOKUP;
5121                 mutex_exit(&drp->r_statelock);
5122         }

5124         *vpp = NULL;
5125         /*
5126          * Lookup this name in the DNLC.  If there is no entry
5127          * lookup over the wire.
5128          */
5129         if (!skipdnlc)
5130                 *vpp = dnlc_lookup(dvp, nm);
5131         if (*vpp == NULL) {
5132                 /*
5133                  * We need to go over the wire to lookup the name.
5134                  */
5135                 return (nfs4lookupnew_otw(dvp, nm, vpp, cr));
5136         }

5138         /*
5139          * We hit on the dnlc
5140          */
```

```
5141         if (*vpp != DNLC_NO_VNODE ||
5142             (dvp->v_vfsp->vfs_flag & VFS_RDONLY)) {
5143                 /*
5144                  * But our attrs may not be valid.
5145                  */
5146                 if (ATTRCACHE4_VALID(dvp)) {
5147                         error = nfs4_waitfor_purge_complete(dvp);
5148                         if (error) {
5149                                 VN_RELE(*vpp);
5150                                 *vpp = NULL;
5151                                 return (error);
5152                         }
5154                         /*
5155                          * If after the purge completes, check to make sure
5156                          * our attrs are still valid.
5157                          */
5158                         if (ATTRCACHE4_VALID(dvp)) {
5159                                 /*
5160                                  * If we waited for a purge we may have
5161                                  * lost our vnode so look it up again.
5162                                  */
5163                                 VN_RELE(*vpp);
5164                                 *vpp = dnlc_lookup(dvp, nm);
5165                                 if (*vpp == NULL)
5166                                         return (nfs4lookupnew_otw(dvp,
5167                                             nm, vpp, cr));
5169                                 /*
5170                                  * The access cache should almost always hit
5171                                  */
5172                                 error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5174                                 if (error) {
5175                                         VN_RELE(*vpp);
5176                                         *vpp = NULL;
5177                                         return (error);
5178                                 }
5179                                 if (*vpp == DNLC_NO_VNODE) {
5180                                         VN_RELE(*vpp);
5181                                         *vpp = NULL;
5182                                         return (ENOENT);
5183                                 }
5184                                 return (0);
5185                         }
5186                 }
5187         }
5189         ASSERT(*vpp != NULL);
5191         /*
5192          * We may have gotten here we have one of the following cases:
5193          *      1) vpp != DNLC_NO_VNODE, our attrs have timed out so we
5194          *              need to validate them.
5195          *      2) vpp == DNLC_NO_VNODE, a negative entry that we always
5196          *              must validate.
5197          *
5198          * Go to the server and check if the directory has changed, if
5199          * it hasn't we are done and can use the dnlc entry.
5200          */
5201         return (nfs4lookupvalidate_otw(dvp, nm, vpp, cr));
5202 }
5204 /*
5205  * Go to the server and check if the directory has changed, if
5206  * it hasn't we are done and can use the dnlc entry.  If it
```



```
5207  * has changed we get a new copy of its attributes and check
5208  * the access for VEXEC, then relookup the filename and
5209  * get its filehandle and attributes.
5210  *
5211  * PUTFH dfh NVERIFY GETATTR ACCESS LOOKUP GETFH GETATTR
5212  *      if the NVERIFY failed we must
5213  *              purge the caches
5214  *              cache new attributes (will set r_time_attr_inval)
5215  *              cache new access
5216  *              recheck VEXEC access
5217  *              add name to dnlc, possibly negative
5218  *              if LOOKUP succeeded
5219  *                      cache new attributes
5220  *      else
5221  *              set a new r_time_attr_inval for dvp
5222  *              check to make sure we have access
5223  *
5224  * The vpp returned is the vnode passed in if the directory is valid,
5225  * a new vnode if successful lookup, or NULL on error.
5226  */
5227 static int
5228 nfs4lookupvalidate_otw(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr)
5229 {
5230         COMPOUND4args_clnt args;
5231         COMPOUND4res_clnt res;
5232         fattr4 *ver_fattr;
5233         fattr4_change dchange;
5234         int32_t *ptr;
5235         int argoplist_size  = 7 * sizeof (nfs_argop4);
5236         nfs_argop4 *argop;
5237         int doqueue;
5238         mntinfo4_t *mi;
5239         nfs4_recov_state_t recov_state;
5240         hrtime_t t;
5241         int isdotdot;
5242         vnode_t *nvp;
5243         nfs_fh4 *fhp;
5244         nfs4_sharedfh_t *sfhp;
5245         nfs4_access_type_t cacc;
5246         rnode4_t *nrp;
5247         rnode4_t *drp = VTOR4(dvp);
5248         nfs4_ga_res_t *garp = NULL;
5249         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
5251         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);
5252         ASSERT(nm != NULL);
5253         ASSERT(nm[0] != '\0');
5254         ASSERT(dvp->v_type == VDIR);
5255         ASSERT(nm[0] != '.' || nm[1] != '\0');
5256         ASSERT(*vpp != NULL);
5258         if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0') {
5259                 isdotdot = 1;
5260                 args.ctag = TAG_LOOKUP_VPARENT;
5261         } else {
5262                 /*
5263                  * If dvp were a stub, it should have triggered and caused
5264                  * a mount for us to get this far.
5265                  */
5266                 ASSERT(!RP_ISSTUB(VTOR4(dvp)));
5268                 isdotdot = 0;
5269                 args.ctag = TAG_LOOKUP_VALID;
5270         }
5272         mi = VTOMI4(dvp);
```

```
5273         recov_state.rs_flags = 0;
5274         recov_state.rs_num_retry_despite_err = 0;

5276         nvp = NULL;

5278         /* Save the original mount point security information */
5279         (void) save_mnt_secinfo(mi->mi_curr_serv);

5281 recov_retry:
5282         e.error = nfs4_start_fop(mi, dvp, NULL, OH_LOOKUP,
5283             &recov_state, NULL);
5284         if (e.error) {
5285                 (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5286                 VN_RELE(*vpp);
5287                 *vpp = NULL;
5288                 return (e.error);
5289         }

5291         argop = kmem_alloc(argoplist_size, KM_SLEEP);

5293         /* PUTFH dfh NVERIFY GETATTR ACCESS LOOKUP GETFH GETATTR */
5294         args.array_len = 7;
5295         args.array = argop;

5297         /* 0. putfh file */
5298         argop[0].argop = OP_CPUTFH;
5299         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(dvp)->r_fh;

5301         /* 1. nverify the change info */
5302         argop[1].argop = OP_NVERIFY;
5303         ver_fattr = &argop[1].nfs_argop4_u.opnverify.obj_attributes;
5304         ver_fattr->attrmask = FATTR4_CHANGE_MASK;
5305         ver_fattr->attrlist4 = (char *)&dchange;
5306         ptr = (int32_t *)&dchange;
5307         IXDR_PUT_HYPER(ptr, VTOR4(dvp)->r_change);
5308         ver_fattr->attrlist4_len = sizeof (fattr4_change);

5310         /* 2. getattr directory */
5311         argop[2].argop = OP_GETATTR;
5312         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5313         argop[2].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5315         /* 3. access directory */
5316         argop[3].argop = OP_ACCESS;
5317         argop[3].nfs_argop4_u.opaccess.access = ACCESS4_READ | ACCESS4_DELETE |
5318             ACCESS4_MODIFY | ACCESS4_EXTEND | ACCESS4_LOOKUP;

5320         /* 4. lookup name */
5321         if (isdotdot) {
5322                 argop[4].argop = OP_LOOKUPP;
5323         } else {
5324                 argop[4].argop = OP_CLOOKUP;
5325                 argop[4].nfs_argop4_u.opclookup.cname = nm;
5326         }

5328         /* 5. resulting file handle */
5329         argop[5].argop = OP_GETFH;

5331         /* 6. resulting file attributes */
5332         argop[6].argop = OP_GETATTR;
5333         argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5334         argop[6].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5336         doqueue = 1;
5337         t = gethrtime();
```



```
5339         rfs4call(VTOMI4(dvp), &args, &res, cr, &doqueue, 0, &e);

5341         if (!isdotdot && res.status == NFS4ERR_MOVED) {
5342                 e.error = nfs4_setup_referral(dvp, nm, vpp, cr);
5343                 if (e.error != 0 && *vpp != NULL)
5344                         VN_RELE(*vpp);
5345                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5346                     &recov_state, FALSE);
5347                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5348                 kmem_free(argop, argoplist_size);
5349                 return (e.error);
5350         }

5352         if (nfs4_needs_recovery(&e, FALSE, dvp->v_vfsp)) {
5353                 /*
5354                  * For WRONGSEC of a non-dotdot case, send secinfo directly
5355                  * from this thread, do not go thru the recovery thread since
5356                  * we need the nm information.
5357                  *
5358                  * Not doing dotdot case because there is no specification
5359                  * for (PUTFH, SECINFO "..") yet.
5360                  */
5361                 if (!isdotdot && res.status == NFS4ERR_WRONGSEC) {
5362                         if ((e.error = nfs4_secinfo_vnode_otw(dvp, nm, cr)))
5363                                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5364                                     &recov_state, FALSE);
5365                         else
5366                                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5367                                     &recov_state, TRUE);
5368                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5369                         kmem_free(argop, argoplist_size);
5370                         if (!e.error)
5371                                 goto recov_retry;
5372                         (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5373                         VN_RELE(*vpp);
5374                         *vpp = NULL;
5375                         return (e.error);
5376                 }

5378                 if (nfs4_start_recovery(&e, mi, dvp, NULL, NULL, NULL,
5379                     OP_LOOKUP, NULL, NULL, NULL) == FALSE) {
5380                         nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5381                             &recov_state, TRUE);

5383                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5384                         kmem_free(argop, argoplist_size);
5385                         goto recov_retry;
5386                 }
5387         }

5389         nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP, &recov_state, FALSE);

5391         if (e.error || res.array_len == 0) {
5392                 /*
5393                  * If e.error isn't set, then reply has no ops (or we couldn't
5394                  * be here).  The only legal way to reply without an op array
5395                  * is via NFS4ERR_MINOR_VERS_MISMATCH.  An ops array should
5396                  * be in the reply for all other status values.
5397                  *
5398                  * For valid replies without an ops array, return ENOTSUP
5399                  * (geterrno4 xlation of VERS_MISMATCH).  For illegal replies,
5400                  * return EIO -- don't trust status.
5401                  */
5402                 if (e.error == 0)
5403                         e.error = (res.status == NFS4ERR_MINOR_VERS_MISMATCH) ?
5404                             ENOTSUP : EIO;
```

```
5405                VN_RELE(*vpp);
5406                *vpp = NULL;
5407                kmem_free(argop, argoplist_size);
5408                (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5409                return (e.error);
5410        }

5412        if (res.status != NFS4ERR_SAME) {
5413                e.error = geterrno4(res.status);

5415                /*
5416                 * The NVERIFY "failed" so the directory has changed
5417                 * First make sure PUTFH succeeded and NVERIFY "failed"
5418                 * cleanly.
5419                 */
5420                if ((res.array[0].nfs_resop4_u.opputfh.status != NFS4_OK) ||
5421                    (res.array[1].nfs_resop4_u.opnverify.status != NFS4_OK)) {
5422                        nfs4_purge_stale_fh(e.error, dvp, cr);
5423                        VN_RELE(*vpp);
5424                        *vpp = NULL;
5425                        goto exit;
5426                }

5428                /*
5429                 * We know the NVERIFY "failed" so we must:
5430                 *      purge the caches (access and indirectly dnlc if needed)
5431                 */
5432                nfs4_purge_caches(dvp, NFS4_NOPURGE_DNLC, cr, TRUE);

5434                if (res.array[2].nfs_resop4_u.opgetattr.status != NFS4_OK) {
5435                        nfs4_purge_stale_fh(e.error, dvp, cr);
5436                        VN_RELE(*vpp);
5437                        *vpp = NULL;
5438                        goto exit;
5439                }

5441                /*
5442                 * Install new cached attributes for the directory
5443                 */
5444                nfs4_attr_cache(dvp,
5445                    &res.array[2].nfs_resop4_u.opgetattr.ga_res,
5446                    t, cr, FALSE, NULL);

5448                if (res.array[3].nfs_resop4_u.opaccess.status != NFS4_OK) {
5449                        nfs4_purge_stale_fh(e.error, dvp, cr);
5450                        VN_RELE(*vpp);
5451                        *vpp = NULL;
5452                        e.error = geterrno4(res.status);
5453                        goto exit;
5454                }

5456                /*
5457                 * Now we know the directory is valid,
5458                 * cache new directory access
5459                 */
5460                nfs4_access_cache(drp,
5461                    args.array[3].nfs_argop4_u.opaccess.access,
5462                    res.array[3].nfs_resop4_u.opaccess.access, cr);

5464                /*
5465                 * recheck VEXEC access
5466                 */
5467                cacc = nfs4_access_check(drp, ACCESS4_LOOKUP, cr);
5468                if (cacc != NFS4_ACCESS_ALLOWED) {
5469                        /*
5470                         * Directory permissions might have been revoked
```



```
5471                         */
5472                        if (cacc == NFS4_ACCESS_DENIED) {
5473                                e.error = EACCES;
5474                                VN_RELE(*vpp);
5475                                *vpp = NULL;
5476                                goto exit;
5477                        }

5479                        /*
5480                         * Somehow we must not have asked for enough
5481                         * so try a singleton ACCESS, should never happen.
5482                         */
5483                        e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5484                        if (e.error) {
5485                                VN_RELE(*vpp);
5486                                *vpp = NULL;
5487                                goto exit;
5488                        }
5489                }

5491                e.error = geterrno4(res.status);
5492                if (res.array[4].nfs_resop4_u.oplookup.status != NFS4_OK) {
5493                        /*
5494                         * The lookup failed, probably no entry
5495                         */
5496                        if (e.error == ENOENT && nfs4_lookup_neg_cache) {
5497                                dnlc_update(dvp, nm, DNLC_NO_VNODE);
5498                        } else {
5499                                /*
5500                                 * Might be some other error, so remove
5501                                 * the dnlc entry to make sure we start all
5502                                 * over again, next time.
5503                                 */
5504                                dnlc_remove(dvp, nm);
5505                        }
5506                        VN_RELE(*vpp);
5507                        *vpp = NULL;
5508                        goto exit;
5509                }

5511                if (res.array[5].nfs_resop4_u.opgetfh.status != NFS4_OK) {
5512                        /*
5513                         * The file exists but we can't get its fh for
5514                         * some unknown reason.  Remove it from the dnlc
5515                         * and error out to be safe.
5516                         */
5517                        dnlc_remove(dvp, nm);
5518                        VN_RELE(*vpp);
5519                        *vpp = NULL;
5520                        goto exit;
5521                }
5522                fhp = &res.array[5].nfs_resop4_u.opgetfh.object;
5523                if (fhp->nfs_fh4_len == 0) {
5524                        /*
5525                         * The file exists but a bogus fh
5526                         * some unknown reason.  Remove it from the dnlc
5527                         * and error out to be safe.
5528                         */
5529                        e.error = ENOENT;
5530                        dnlc_remove(dvp, nm);
5531                        VN_RELE(*vpp);
5532                        *vpp = NULL;
5533                        goto exit;
5534                }
5535                sfhp = sfh4_get(fhp, mi);
```

```
5537                if (res.array[6].nfs_resop4_u.opgetattr.status == NFS4_OK)
5538                        garp = &res.array[6].nfs_resop4_u.opgetattr.ga_res;
5539
5540                /*
5541                 * Make the new rnode
5542                 */
5543                if (isdotdot) {
5544                        e.error = nfs4_make_dotdot(sfhp, t, dvp, cr, &nvp, 1);
5545                        if (e.error) {
5546                                sfh4_rele(&sfhp);
5547                                VN_RELE(*vpp);
5548                                *vpp = NULL;
5549                                goto exit;
5550                        }
5551                        /*
5552                         * XXX if nfs4_make_dotdot uses an existing rnode
5553                         * XXX it doesn't update the attributes.
5554                         * XXX for now just save them again to save an OTW
5555                         */
5556                        nfs4_attr_cache(nvp, garp, t, cr, FALSE, NULL);
5557                } else {
5558                        nvp = makenfs4node(sfhp, garp, dvp->v_vfsp, t, cr,
5559                            dvp, fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
5560                        /*
5561                         * If v_type == VNON, then garp was NULL because
5562                         * the last op in the compound failed and makenfs4node
5563                         * could not find the vnode for sfhp. It created
5564                         * a new vnode, so we have nothing to purge here.
5565                         */
5566                        if (nvp->v_type == VNON) {
5567                                vattr_t vattr;
5568
5569                                vattr.va_mask = AT_TYPE;
5570                                /*
5571                                 * N.B. We've already called nfs4_end_fop above.
5572                                 */
5573                                e.error = nfs4getattr(nvp, &vattr, cr);
5574                                if (e.error) {
5575                                        sfh4_rele(&sfhp);
5576                                        VN_RELE(*vpp);
5577                                        *vpp = NULL;
5578                                        VN_RELE(nvp);
5579                                        goto exit;
5580                                }
5581                                nvp->v_type = vattr.va_type;
5582                        }
5583                }
5584                sfh4_rele(&sfhp);
5585
5586                nrp = VTOR4(nvp);
5587                mutex_enter(&nrp->r_statev4_lock);
5588                if (!nrp->created_v4) {
5589                        mutex_exit(&nrp->r_statev4_lock);
5590                        dnlc_update(dvp, nm, nvp);
5591                } else
5592                        mutex_exit(&nrp->r_statev4_lock);
5593
5594                VN_RELE(*vpp);
5595                *vpp = nvp;
5596        } else {
5597                hrtime_t now;
5598                hrtime_t delta = 0;
5599
5600                e.error = 0;
5601
5602                /*
```



```
5603                 * Because the NVERIFY "succeeded" we know that the
5604                 * directory attributes are still valid
5605                 * so update r_time_attr_inval
5606                 */
5607                now = gethrtime();
5608                mutex_enter(&drp->r_statelock);
5609                if (!(mi->mi_flags & MI4_NOAC) && !(dvp->v_flag & VNOCACHE)) {
5610                        delta = now - drp->r_time_attr_saved;
5611                        if (delta < mi->mi_acdirmin)
5612                                delta = mi->mi_acdirmin;
5613                        else if (delta > mi->mi_acdirmax)
5614                                delta = mi->mi_acdirmax;
5615                }
5616                drp->r_time_attr_inval = now + delta;
5617                mutex_exit(&drp->r_statelock);
5618                dnlc_update(dvp, nm, *vpp);
5619
5620                /*
5621                 * Even though we have a valid directory attr cache
5622                 * and dnlc entry, we may not have access.
5623                 * This should almost always hit the cache.
5624                 */
5625                e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5626                if (e.error) {
5627                        VN_RELE(*vpp);
5628                        *vpp = NULL;
5629                }
5630
5631                if (*vpp == DNLC_NO_VNODE) {
5632                        VN_RELE(*vpp);
5633                        *vpp = NULL;
5634                        e.error = ENOENT;
5635                }
5636        }
5637
5638 exit:
5639        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5640        kmem_free(argop, argoplist_size);
5641        (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5642        return (e.error);
5643 }
5644
5645 /*
5646  * We need to go over the wire to lookup the name, but
5647  * while we are there verify the directory has not
5648  * changed but if it has, get new attributes and check access
5649  *
5650  * PUTFH dfh SAVEFH LOOKUP nm GETFH GETATTR RESTOREFH
5651  *                                      NVERIFY GETATTR ACCESS
5652  *
5653  * With the results:
5654  *      if the NVERIFY failed we must purge the caches, add new attributes,
5655  *              and cache new access.
5656  *      set a new r_time_attr_inval
5657  *      add name to dnlc, possibly negative
5658  *      if LOOKUP succeeded
5659  *              cache new attributes
5660  */
5661 static int
5662 nfs4lookupnew_otw(vnode_t *dvp, char *nm, vnode_t **vpp, cred_t *cr)
5663 {
5664        COMPOUND4args_clnt args;
5665        COMPOUND4res_clnt res;
5666        fattr4 *ver_fattr;
5667        fattr4_change dchange;
5668        int32_t *ptr;
```

```
5669         nfs4_ga_res_t *garp = NULL;
5670         int argoplist_size  = 9 * sizeof (nfs_argop4);
5671         nfs_argop4 *argop;
5672         int doqueue;
5673         mntinfo4_t *mi;
5674         nfs4_recov_state_t recov_state;
5675         hrtime_t t;
5676         int isdotdot;
5677         vnode_t *nvp;
5678         nfs_fh4 *fhp;
5679         nfs4_sharedfh_t *sfhp;
5680         nfs4_access_type_t cacc;
5681         rnode4_t *nrp;
5682         rnode4_t *drp = VTOR4(dvp);
5683         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

5685         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);
5686         ASSERT(nm != NULL);
5687         ASSERT(nm[0] != '\0');
5688         ASSERT(dvp->v_type == VDIR);
5689         ASSERT(nm[0] != '.' || nm[1] != '\0');
5690         ASSERT(*vpp == NULL);

5692         if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0') {
5693                 isdotdot = 1;
5694                 args.ctag = TAG_LOOKUP_PARENT;
5695         } else {
5696                 /*
5697                  * If dvp were a stub, it should have triggered and caused
5698                  * a mount for us to get this far.
5699                  */
5700                 ASSERT(!RP_ISSTUB(VTOR4(dvp)));

5702                 isdotdot = 0;
5703                 args.ctag = TAG_LOOKUP;
5704         }

5706         mi = VTOMI4(dvp);
5707         recov_state.rs_flags = 0;
5708         recov_state.rs_num_retry_despite_err = 0;

5710         nvp = NULL;

5712         /* Save the original mount point security information */
5713         (void) save_mnt_secinfo(mi->mi_curr_serv);

5715 recov_retry:
5716         e.error = nfs4_start_fop(mi, dvp, NULL, OH_LOOKUP,
5717             &recov_state, NULL);
5718         if (e.error) {
5719                 (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5720                 return (e.error);
5721         }

5723         argop = kmem_alloc(argoplist_size, KM_SLEEP);

5725         /* PUTFH SAVEFH LOOKUP GETFH GETATTR RESTOREFH NVERIFY GETATTR ACCESS */
5726         args.array_len = 9;
5727         args.array = argop;

5729         /* 0. putfh file */
5730         argop[0].argop = OP_CPUTFH;
5731         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(dvp)->r_fh;

5733         /* 1. savefh for the nverify */
5734         argop[1].argop = OP_SAVEFH;
```



```
5736         /* 2. lookup name */
5737         if (isdotdot) {
5738                 argop[2].argop = OP_LOOKUPP;
5739         } else {
5740                 argop[2].argop = OP_CLOOKUP;
5741                 argop[2].nfs_argop4_u.opclookup.cname = nm;
5742         }

5744         /* 3. resulting file handle */
5745         argop[3].argop = OP_GETFH;

5747         /* 4. resulting file attributes */
5748         argop[4].argop = OP_GETATTR;
5749         argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5750         argop[4].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5752         /* 5. restorefh back the directory for the nverify */
5753         argop[5].argop = OP_RESTOREFH;

5755         /* 6. nverify the change info */
5756         argop[6].argop = OP_NVERIFY;
5757         ver_fattr = &argop[6].nfs_argop4_u.opnverify.obj_attributes;
5758         ver_fattr->attrmask = FATTR4_CHANGE_MASK;
5759         ver_fattr->attrlist4 = (char *)&dchange;
5760         ptr = (int32_t *)&dchange;
5761         IXDR_PUT_HYPER(ptr, VTOR4(dvp)->r_change);
5762         ver_fattr->attrlist4_len = sizeof (fattr4_change);

5764         /* 7. getattr directory */
5765         argop[7].argop = OP_GETATTR;
5766         argop[7].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
5767         argop[7].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

5769         /* 8. access directory */
5770         argop[8].argop = OP_ACCESS;
5771         argop[8].nfs_argop4_u.opaccess.access = ACCESS4_READ | ACCESS4_DELETE |
5772             ACCESS4_MODIFY | ACCESS4_EXTEND | ACCESS4_LOOKUP;

5774         doqueue = 1;
5775         t = gethrtime();

5777         rfs4call(VTOMI4(dvp), &args, &res, cr, &doqueue, 0, &e);

5779         if (!isdotdot && res.status == NFS4ERR_MOVED) {
5780                 e.error = nfs4_setup_referral(dvp, nm, vpp, cr);
5781                 if (e.error != 0 && *vpp != NULL)
5782                         VN_RELE(*vpp);
5783                 nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5784                     &recov_state, FALSE);
5785                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5786                 kmem_free(argop, argoplist_size);
5787                 return (e.error);
5788         }

5790         if (nfs4_needs_recovery(&e, FALSE, dvp->v_vfsp)) {
5791                 /*
5792                  * For WRONGSEC of a non-dotdot case, send secinfo directly
5793                  * from this thread, do not go thru the recovery thread since
5794                  * we need the nm information.
5795                  *
5796                  * Not doing dotdot case because there is no specification
5797                  * for (PUTFH, SECINFO "..") yet.
5798                  */
5799                 if (!isdotdot && res.status == NFS4ERR_WRONGSEC) {
5800                         if ((e.error = nfs4_secinfo_vnode_otw(dvp, nm, cr)))
```

```
5801                                nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5802                                    &recov_state, FALSE);
5803                        else
5804                                nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5805                                    &recov_state, TRUE);
5806                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5807                        kmem_free(argop, argoplist_size);
5808                        if (!e.error)
5809                                goto recov_retry;
5810                        (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5811                        return (e.error);
5812                }

5814                if (nfs4_start_recovery(&e, mi, dvp, NULL, NULL, NULL,
5815                    OP_LOOKUP, NULL, NULL, NULL) == FALSE) {
5816                        nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP,
5817                            &recov_state, TRUE);

5819                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
5820                        kmem_free(argop, argoplist_size);
5821                        goto recov_retry;
5822                }
5823        }

5825        nfs4_end_fop(mi, dvp, NULL, OH_LOOKUP, &recov_state, FALSE);

5827        if (e.error || res.array_len == 0) {
5828                /*
5829                 * If e.error isn't set, then reply has no ops (or we couldn't
5830                 * be here).  The only legal way to reply without an op array
5831                 * is via NFS4ERR_MINOR_VERS_MISMATCH.  An ops array should
5832                 * be in the reply for all other status values.
5833                 *
5834                 * For valid replies without an ops array, return ENOTSUP
5835                 * (geterrno4 xlation of VERS_MISMATCH).  For illegal replies,
5836                 * return EIO -- don't trust status.
5837                 */
5838                if (e.error == 0)
5839                        e.error = (res.status == NFS4ERR_MINOR_VERS_MISMATCH) ?
5840                            ENOTSUP : EIO;

5842                kmem_free(argop, argoplist_size);
5843                (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
5844                return (e.error);
5845        }

5847        e.error = geterrno4(res.status);

5849        /*
5850         * The PUTFH and SAVEFH may have failed.
5851         */
5852        if ((res.array[0].nfs_resop4_u.opputfh.status != NFS4_OK) ||
5853            (res.array[1].nfs_resop4_u.opsavefh.status != NFS4_OK)) {
5854                nfs4_purge_stale_fh(e.error, dvp, cr);
5855                goto exit;
5856        }

5858        /*
5859         * Check if the file exists, if it does delay entering
5860         * into the dnlc until after we update the directory
5861         * attributes so we don't cause it to get purged immediately.
5862         */
5863        if (res.array[2].nfs_resop4_u.oplookup.status != NFS4_OK) {
5864                /*
5865                 * The lookup failed, probably no entry
5866                 */
```



```
5867                if (e.error == ENOENT && nfs4_lookup_neg_cache)
5868                        dnlc_update(dvp, nm, DNLC_NO_VNODE);
5869                goto exit;
5870        }

5872        if (res.array[3].nfs_resop4_u.opgetfh.status != NFS4_OK) {
5873                /*
5874                 * The file exists but we can't get its fh for
5875                 * some unknown reason. Error out to be safe.
5876                 */
5877                goto exit;
5878        }

5880        fhp = &res.array[3].nfs_resop4_u.opgetfh.object;
5881        if (fhp->nfs_fh4_len == 0) {
5882                /*
5883                 * The file exists but a bogus fh
5884                 * some unknown reason.  Error out to be safe.
5885                 */
5886                e.error = EIO;
5887                goto exit;
5888        }
5889        sfhp = sfh4_get(fhp, mi);

5891        if (res.array[4].nfs_resop4_u.opgetattr.status != NFS4_OK) {
5892                sfh4_rele(&sfhp);
5893                goto exit;
5894        }
5895        garp = &res.array[4].nfs_resop4_u.opgetattr.ga_res;

5897        /*
5898         * The RESTOREFH may have failed
5899         */
5900        if (res.array[5].nfs_resop4_u.oprestorefh.status != NFS4_OK) {
5901                sfh4_rele(&sfhp);
5902                e.error = EIO;
5903                goto exit;
5904        }

5906        if (res.array[6].nfs_resop4_u.opnverify.status != NFS4ERR_SAME) {
5907                /*
5908                 * First make sure the NVERIFY failed as we expected,
5909                 * if it didn't then be conservative and error out
5910                 * as we can't trust the directory.
5911                 */
5912                if (res.array[6].nfs_resop4_u.opnverify.status != NFS4_OK) {
5913                        sfh4_rele(&sfhp);
5914                        e.error = EIO;
5915                        goto exit;
5916                }

5918                /*
5919                 * We know the NVERIFY "failed" so the directory has changed,
5920                 * so we must:
5921                 *      purge the caches (access and indirectly dnlc if needed)
5922                 */
5923                nfs4_purge_caches(dvp, NFS4_NOPURGE_DNLC, cr, TRUE);

5925                if (res.array[7].nfs_resop4_u.opgetattr.status != NFS4_OK) {
5926                        sfh4_rele(&sfhp);
5927                        goto exit;
5928                }
5929                nfs4_attr_cache(dvp,
5930                    &res.array[7].nfs_resop4_u.opgetattr.ga_res,
5931                    t, cr, FALSE, NULL);
```

```
5933                if (res.array[8].nfs_resop4_u.opaccess.status != NFS4_OK) {
5934                        nfs4_purge_stale_fh(e.error, dvp, cr);
5935                        sfh4_rele(&sfhp);
5936                        e.error = geterrno4(res.status);
5937                        goto exit;
5938                }

5940                /*
5941                 * Now we know the directory is valid,
5942                 * cache new directory access
5943                 */
5944                nfs4_access_cache(drp,
5945                    args.array[8].nfs_argop4_u.opaccess.access,
5946                    res.array[8].nfs_resop4_u.opaccess.access, cr);

5948                /*
5949                 * recheck VEXEC access
5950                 */
5951                cacc = nfs4_access_check(drp, ACCESS4_LOOKUP, cr);
5952                if (cacc != NFS4_ACCESS_ALLOWED) {
5953                        /*
5954                         * Directory permissions might have been revoked
5955                         */
5956                        if (cacc == NFS4_ACCESS_DENIED) {
5957                                sfh4_rele(&sfhp);
5958                                e.error = EACCES;
5959                                goto exit;
5960                        }

5962                        /*
5963                         * Somehow we must not have asked for enough
5964                         * so try a singleton ACCESS should never happen
5965                         */
5966                        e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
5967                        if (e.error) {
5968                                sfh4_rele(&sfhp);
5969                                goto exit;
5970                        }
5971                }

5973                e.error = geterrno4(res.status);
5974        } else {
5975                hrtime_t now;
5976                hrtime_t delta = 0;

5978                e.error = 0;

5980                /*
5981                 * Because the NVERIFY "succeeded" we know that the
5982                 * directory attributes are still valid
5983                 * so update r_time_attr_inval
5984                 */
5985                now = gethrtime();
5986                mutex_enter(&drp->r_statelock);
5987                if (!(mi->mi_flags & MI4_NOAC) && !(dvp->v_flag & VNOCACHE)) {
5988                        delta = now - drp->r_time_attr_saved;
5989                        if (delta < mi->mi_acdirmin)
5990                                delta = mi->mi_acdirmin;
5991                        else if (delta > mi->mi_acdirmax)
5992                                delta = mi->mi_acdirmax;
5993                }
5994                drp->r_time_attr_inval = now + delta;
5995                mutex_exit(&drp->r_statelock);

5997                /*
5998                 * Even though we have a valid directory attr cache,
```



```
5999                 * we may not have access.
6000                 * This should almost always hit the cache.
6001                 */
6002                e.error = nfs4_access(dvp, VEXEC, 0, cr, NULL);
6003                if (e.error) {
6004                        sfh4_rele(&sfhp);
6005                        goto exit;
6006                }
6007        }

6009        /*
6010         * Now we have successfully completed the lookup, if the
6011         * directory has changed we now have the valid attributes.
6012         * We also know we have directory access.
6013         * Create the new rnode and insert it in the dnlc.
6014         */
6015        if (isdotdot) {
6016                e.error = nfs4_make_dotdot(sfhp, t, dvp, cr, &nvp, 1);
6017                if (e.error) {
6018                        sfh4_rele(&sfhp);
6019                        goto exit;
6020                }
6021                /*
6022                 * XXX if nfs4_make_dotdot uses an existing rnode
6023                 * XXX it doesn't update the attributes.
6024                 * XXX for now just save them again to save an OTW
6025                 */
6026                nfs4_attr_cache(nvp, garp, t, cr, FALSE, NULL);
6027        } else {
6028                nvp = makenfs4node(sfhp, garp, dvp->v_vfsp, t, cr,
6029                    dvp, fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
6030        }
6031        sfh4_rele(&sfhp);

6033        nrp = VTOR4(nvp);
6034        mutex_enter(&nrp->r_statev4_lock);
6035        if (!nrp->created_v4) {
6036                mutex_exit(&nrp->r_statev4_lock);
6037                dnlc_update(dvp, nm, nvp);
6038        } else
6039                mutex_exit(&nrp->r_statev4_lock);

6041        *vpp = nvp;

6043 exit:
6044        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6045        kmem_free(argop, argoplist_size);
6046        (void) check_mnt_secinfo(mi->mi_curr_serv, nvp);
6047        return (e.error);
6048 }

6050 #ifdef DEBUG
6051 void
6052 nfs4lookup_dump_compound(char *where, nfs_argop4 *argbase, int argcnt)
6053 {
6054        uint_t i, len;
6055        zoneid_t zoneid = getzoneid();
6056        char *s;

6058        zcmn_err(zoneid, CE_NOTE, "%s: dumping cmpd", where);
6059        for (i = 0; i < argcnt; i++) {
6060                nfs_argop4 *op = &argbase[i];
6061                switch (op->argop) {
6062                case OP_CPUTFH:
6063                case OP_PUTFH:
6064                        zcmn_err(zoneid, CE_NOTE, "\t op %d, putfh", i);
```

```
6065                        break;
6066                case OP_PUTROOTFH:
6067                        zcmn_err(zoneid, CE_NOTE, "\t op %d, putrootfh", i);
6068                        break;
6069                case OP_CLOOKUP:
6070                        s = op->nfs_argop4_u.opclookup.cname;
6071                        zcmn_err(zoneid, CE_NOTE, "\t op %d, lookup %s", i, s);
6072                        break;
6073                case OP_LOOKUP:
6074                        s = utf8_to_str(&op->nfs_argop4_u.oplookup.objname,
6075                            &len, NULL);
6076                        zcmn_err(zoneid, CE_NOTE, "\t op %d, lookup %s", i, s);
6077                        kmem_free(s, len);
6078                        break;
6079                case OP_LOOKUPP:
6080                        zcmn_err(zoneid, CE_NOTE, "\t op %d, lookupp ..", i);
6081                        break;
6082                case OP_GETFH:
6083                        zcmn_err(zoneid, CE_NOTE, "\t op %d, getfh", i);
6084                        break;
6085                case OP_GETATTR:
6086                        zcmn_err(zoneid, CE_NOTE, "\t op %d, getattr", i);
6087                        break;
6088                case OP_OPENATTR:
6089                        zcmn_err(zoneid, CE_NOTE, "\t op %d, openattr", i);
6090                        break;
6091                default:
6092                        zcmn_err(zoneid, CE_NOTE, "\t op %d, opcode %d", i,
6093                            op->argop);
6094                        break;
6095                }
6096        }
6097 }
6098 #endif

6100 /*
6101  * nfs4lookup_setup - constructs a multi-lookup compound request.
6102  *
6103  * Given the path "nm1/nm2/.../nmn", the following compound requests
6104  * may be created:
6105  *
6106  * Note: Getfh is not be needed because filehandle attr is mandatory, but it
6107  * is faster, for now.
6108  *
6109  * l4_getattrs indicates the type of compound requested.
6110  *
6111  * LKP4_NO_ATTRIBUTE - no attributes (used by secinfo):
6112  *
6113  *      compound { Put*fh; Lookup {nm1}; Lookup {nm2}; ...  Lookup {nmn} }
6114  *
6115  *   total number of ops is n + 1.
6116  *
6117  * LKP4_LAST_NAMED_ATTR - multi-component path for a named
6118  *      attribute: create lookups plus one OPENATTR/GETFH/GETATTR
6119  *      before the last component, and only get attributes
6120  *      for the last component.  Note that the second-to-last
6121  *      pathname component is XATTR_RPATH, which does NOT go
6122  *      over-the-wire as a lookup.
6123  *
6124  *      compound { Put*fh; Lookup {nm1}; Lookup {nm2}; ... Lookup {nmn-2};
6125  *              Openattr; Getfh; Getattr; Lookup {nmn}; Getfh; Getattr }
6126  *
6127  *   and total number of ops is n + 5.
6128  *
6129  * LKP4_LAST_ATTRDIR - multi-component path for the hidden named
6130  *      attribute directory: create lookups plus an OPENATTR
```



```
6131  *      replacing the last lookup.  Note that the last pathname
6132  *      component is XATTR_RPATH, which does NOT go over-the-wire
6133  *      as a lookup.
6134  *
6135  *      compound { Put*fh; Lookup {nm1}; Lookup {nm2}; ... Getfh; Getattr;
6136  *              Openattr; Getfh; Getattr }
6137  *
6138  *   and total number of ops is n + 5.
6139  *
6140  * LKP4_ALL_ATTRIBUTES - create lookups and get attributes for intermediate
6141  *      nodes too.
6142  *
6143  *      compound { Put*fh; Lookup {nm1}; Getfh; Getattr;
6144  *              Lookup {nm2}; ...  Lookup {nmn}; Getfh; Getattr }
6145  *
6146  *   and total number of ops is 3*n + 1.
6147  *
6148  * All cases: returns the index in the arg array of the final LOOKUP op, or
6149  * -1 if no LOOKUPs were used.
6150  */
6151 int
6152 nfs4lookup_setup(char *nm, lookup4_param_t *lookupargp, int needgetfh)
6153 {
6154         enum lkp4_attr_setup l4_getattrs = lookupargp->l4_getattrs;
6155         nfs_argop4 *argbase, *argop;
6156         int arglen, argcnt;
6157         int n = 1;      /* number of components */
6158         int nga = 1;    /* number of Getattr's in request */
6159         char c = '\0', *s, *p;
6160         int lookup_idx = -1;
6161         int argoplist_size;

6163         /* set lookuparg response result to 0 */
6164         lookupargp->resp->status = NFS4_OK;

6166         /* skip leading "/" or "." e.g. ".//./" if there is */
6167         for (; ; nm++) {
6168                 if (*nm != '/' && *nm != '.')
6169                         break;

6171                 /* ".." is counted as 1 component */
6172                 if (*nm == '.' && *(nm + 1) != '/')
6173                         break;
6174         }

6176         /*
6177          * Find n = number of components - nm must be null terminated
6178          * Skip "." components.
6179          */
6180         if (*nm != '\0')
6181                 for (n = 1, s = nm; *s != '\0'; s++) {
6182                         if ((*s == '/') && (*(s + 1) != '/') &&
6183                             (*(s + 1) != '\0') &&
6184                             !(*(s + 1) == '.' && (*(s + 2) == '/' ||
6185                             *(s + 2) == '\0')))
6186                                 n++;
6187                 }
6188         else
6189                 n = 0;

6191         /*
6192          * nga is number of components that need Getfh+Getattr
6193          */
6194         switch (l4_getattrs) {
6195         case LKP4_NO_ATTRIBUTES:
6196                 nga = 0;
```

```
6197                    break;
6198            case LKP4_ALL_ATTRIBUTES:
6199                    nga = n;
6200                    /*
6201                     * Always have at least 1 getfh, getattr pair
6202                     */
6203                    if (nga == 0)
6204                            nga++;
6205                    break;
6206            case LKP4_LAST_ATTRDIR:
6207            case LKP4_LAST_NAMED_ATTR:
6208                    nga = n+1;
6209                    break;
6210            }

6212            /*
6213             * If change to use the filehandle attr instead of getfh
6214             * the following line can be deleted.
6215             */
6216            nga *= 2;

6218            /*
6219             * calculate number of ops in request as
6220             * header + trailer + lookups + getattrs
6221             */
6222            arglen = lookupargp->header_len + lookupargp->trailer_len + n + nga;

6224            argoplist_size = arglen * sizeof (nfs_argop4);
6225            argop = argbase = kmem_alloc(argoplist_size, KM_SLEEP);
6226            lookupargp->argsp->array = argop;

6228            argcnt = lookupargp->header_len;
6229            argop += argcnt;

6231            /*
6232             * loop and create a lookup op and possibly getattr/getfh for
6233             * each component. Skip "." components.
6234             */
6235            for (s = nm; *s != '\0'; s = p) {
6236                    /*
6237                     * Set up a pathname struct for each component if needed
6238                     */
6239                    while (*s == '/')
6240                            s++;
6241                    if (*s == '\0')
6242                            break;

6244                    for (p = s; (*p != '/') && (*p != '\0'); p++)
6245                            ;
6246                    c = *p;
6247                    *p = '\0';

6249                    if (s[0] == '.' && s[1] == '\0') {
6250                            *p = c;
6251                            continue;
6252                    }
6253                    if (l4_getattrs == LKP4_LAST_ATTRDIR &&
6254                        strcmp(s, XATTR_RPATH) == 0) {
6255                            /* getfh XXX may not be needed in future */
6256                            argop->argop = OP_GETFH;
6257                            argop++;
6258                            argcnt++;

6260                            /* getattr */
6261                            argop->argop = OP_GETATTR;
6262                            argop->nfs_argop4_u.opgetattr.attr_request =
```



```
6263                                lookupargp->ga_bits;
6264                            argop->nfs_argop4_u.opgetattr.mi =
6265                                lookupargp->mi;
6266                            argop++;
6267                            argcnt++;

6269                            /* openattr */
6270                            argop->argop = OP_OPENATTR;
6271                    } else if (l4_getattrs == LKP4_LAST_NAMED_ATTR &&
6272                        strcmp(s, XATTR_RPATH) == 0) {
6273                            /* openattr */
6274                            argop->argop = OP_OPENATTR;
6275                            argop++;
6276                            argcnt++;

6278                            /* getfh XXX may not be needed in future */
6279                            argop->argop = OP_GETFH;
6280                            argop++;
6281                            argcnt++;

6283                            /* getattr */
6284                            argop->argop = OP_GETATTR;
6285                            argop->nfs_argop4_u.opgetattr.attr_request =
6286                                lookupargp->ga_bits;
6287                            argop->nfs_argop4_u.opgetattr.mi =
6288                                lookupargp->mi;
6289                            argop++;
6290                            argcnt++;
6291                            *p = c;
6292                            continue;
6293                    } else if (s[0] == '.' && s[1] == '.' && s[2] == '\0') {
6294                            /* lookupp */
6295                            argop->argop = OP_LOOKUPP;
6296                    } else {
6297                            /* lookup */
6298                            argop->argop = OP_LOOKUP;
6299                            (void) str_to_utf8(s,
6300                                &argop->nfs_argop4_u.oplookup.objname);
6301                    }
6302                    lookup_idx = argcnt;
6303                    argop++;
6304                    argcnt++;

6306                    *p = c;

6308                    if (l4_getattrs == LKP4_ALL_ATTRIBUTES) {
6309                            /* getfh XXX may not be needed in future */
6310                            argop->argop = OP_GETFH;
6311                            argop++;
6312                            argcnt++;

6314                            /* getattr */
6315                            argop->argop = OP_GETATTR;
6316                            argop->nfs_argop4_u.opgetattr.attr_request =
6317                                lookupargp->ga_bits;
6318                            argop->nfs_argop4_u.opgetattr.mi =
6319                                lookupargp->mi;
6320                            argop++;
6321                            argcnt++;
6322                    }
6323            }

6325            if ((l4_getattrs != LKP4_NO_ATTRIBUTES) &&
6326                ((l4_getattrs != LKP4_ALL_ATTRIBUTES) || (lookup_idx < 0))) {
6327                    if (needgetfh) {
6328                            /* stick in a post-lookup getfh */
```

```
6329                        argop->argop = OP_GETFH;
6330                        argcnt++;
6331                        argop++;
6332                }
6333                /* post-lookup getattr */
6334                argop->argop = OP_GETATTR;
6335                argop->nfs_argop4_u.opgetattr.attr_request =
6336                    lookupargp->ga_bits;
6337                argop->nfs_argop4_u.opgetattr.mi = lookupargp->mi;
6338                argcnt++;
6339        }
6340        argcnt += lookupargp->trailer_len;      /* actual op count */
6341        lookupargp->argsp->array_len = argcnt;
6342        lookupargp->arglen = arglen;

6344 #ifdef DEBUG
6345        if (nfs4_client_lookup_debug)
6346                nfs4lookup_dump_compound("nfs4lookup_setup", argbase, argcnt);
6347 #endif

6349        return (lookup_idx);
6350 }

6352 static int
6353 nfs4openattr(vnode_t *dvp, vnode_t **avp, int cflag, cred_t *cr)
6354 {
6355        COMPOUND4args_clnt      args;
6356        COMPOUND4res_clnt       res;
6357        GETFH4res       *gf_res = NULL;
6358        nfs_argop4      argop[4];
6359        nfs_resop4      *resop = NULL;
6360        nfs4_sharedfh_t *sfhp;
6361        hrtime_t t;
6362        nfs4_error_t    e;

6364        rnode4_t        *drp;
6365        int             doqueue = 1;
6366        vnode_t         *vp;
6367        int             needrecov = 0;
6368        nfs4_recov_state_t recov_state;

6370        ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);

6372        *avp = NULL;
6373        recov_state.rs_flags = 0;
6374        recov_state.rs_num_retry_despite_err = 0;

6376 recov_retry:
6377        /* COMPOUND: putfh, openattr, getfh, getattr */
6378        args.array_len = 4;
6379        args.array = argop;
6380        args.ctag = TAG_OPENATTR;

6382        e.error = nfs4_start_op(VTOMI4(dvp), dvp, NULL, &recov_state);
6383        if (e.error)
6384                return (e.error);

6386        drp = VTOR4(dvp);

6388        /* putfh */
6389        argop[0].argop = OP_CPUTFH;
6390        argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

6392        /* openattr */
6393        argop[1].argop = OP_OPENATTR;
6394        argop[1].nfs_argop4_u.opopenattr.createdir = (cflag ? TRUE : FALSE);
```



```
6396        /* getfh */
6397        argop[2].argop = OP_GETFH;

6399        /* getattr */
6400        argop[3].argop = OP_GETATTR;
6401        argop[3].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
6402        argop[3].nfs_argop4_u.opgetattr.mi = VTOMI4(dvp);

6404        NFS4_DEBUG(nfs4_client_call_debug, (CE_NOTE,
6405            "nfs4openattr: %s call, drp %s", needrecov ? "recov" : "first",
6406            rnode4info(drp)));

6408        t = gethrtime();

6410        rfs4call(VTOMI4(dvp), &args, &res, cr, &doqueue, 0, &e);

6412        needrecov = nfs4_needs_recovery(&e, FALSE, dvp->v_vfsp);
6413        if (needrecov) {
6414                bool_t abort;

6416                NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
6417                    "nfs4openattr: initiating recovery\n"));

6419                abort = nfs4_start_recovery(&e,
6420                    VTOMI4(dvp), dvp, NULL, NULL, NULL,
6421                    OP_OPENATTR, NULL, NULL, NULL);
6422                nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);
6423                if (!e.error) {
6424                        e.error = geterrno4(res.status);
6425                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6426                }
6427                if (abort == FALSE)
6428                        goto recov_retry;
6429                return (e.error);
6430        }

6432        if (e.error) {
6433                nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);
6434                return (e.error);
6435        }

6437        if (res.status) {
6438                /*
6439                 * If OTW errro is NOTSUPP, then it should be
6440                 * translated to EINVAL.  All Solaris file system
6441                 * implementations return EINVAL to the syscall layer
6442                 * when the attrdir cannot be created due to an
6443                 * implementation restriction or noxattr mount option.
6444                 */
6445                if (res.status == NFS4ERR_NOTSUPP) {
6446                        mutex_enter(&drp->r_statelock);
6447                        if (drp->r_xattr_dir)
6448                                VN_RELE(drp->r_xattr_dir);
6449                        VN_HOLD(NFS4_XATTR_DIR_NOTSUPP);
6450                        drp->r_xattr_dir = NFS4_XATTR_DIR_NOTSUPP;
6451                        mutex_exit(&drp->r_statelock);

6453                        e.error = EINVAL;
6454                } else {
6455                        e.error = geterrno4(res.status);
6456                }

6458                if (e.error) {
6459                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6460                        nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state,
```

```
6461                            needrecov);
6462                        return (e.error);
6463                }
6464        }

6466        resop = &res.array[0];  /* putfh res */
6467        ASSERT(resop->nfs_resop4_u.opgetfh.status == NFS4_OK);

6469        resop = &res.array[1];  /* openattr res */
6470        ASSERT(resop->nfs_resop4_u.opopenattr.status == NFS4_OK);

6472        resop = &res.array[2];  /* getfh res */
6473        gf_res = &resop->nfs_resop4_u.opgetfh;
6474        if (gf_res->object.nfs_fh4_len == 0) {
6475                *avp = NULL;
6476                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
6477                nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);
6478                return (ENOENT);
6479        }

6481        sfhp = sfh4_get(&gf_res->object, VTOMI4(dvp));
6482        vp = makenfs4node(sfhp, &res.array[3].nfs_resop4_u.opgetattr.ga_res,
6483            dvp->v_vfsp, t, cr, dvp,
6484            fn_get(VTOSV(dvp)->sv_name, XATTR_RPATH, sfhp));
6485        sfh4_rele(&sfhp);

6487        if (e.error)
6488                PURGE_ATTRCACHE4(vp);

6490        mutex_enter(&vp->v_lock);
6491        vp->v_flag |= V_XATTRDIR;
6492        mutex_exit(&vp->v_lock);

6494        *avp = vp;

6496        mutex_enter(&drp->r_statelock);
6497        if (drp->r_xattr_dir)
6498                VN_RELE(drp->r_xattr_dir);
6499        VN_HOLD(vp);
6500        drp->r_xattr_dir = vp;

6502        /*
6503         * Invalidate pathconf4 cache because r_xattr_dir is no longer
6504         * NULL.  xattrs could be created at any time, and we have no
6505         * way to update pc4_xattr_exists in the base object if/when
6506         * it happens.
6507         */
6508        drp->r_pathconf.pc4_xattr_valid = 0;

6510        mutex_exit(&drp->r_statelock);

6512        nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);

6514        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

6516        return (0);
6517 }

6519 /* ARGSUSED */
6520 static int
6521 nfs4_create(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
6522         int mode, vnode_t **vpp, cred_t *cr, int flags, caller_context_t *ct,
6523         vsecattr_t *vsecp)
6524 {
6525         int error;
6526         vnode_t *vp = NULL;
```



```
6527         rnode4_t *rp;
6528         struct vattr vattr;
6529         rnode4_t *drp;
6530         vnode_t *tempvp;
6531         enum createmode4 createmode;
6532         bool_t must_trunc = FALSE;
6533         int     truncating = 0;

6535         if (nfs_zone() != VTOMI4(dvp)->mi_zone)
6536                 return (EPERM);
6537         if (exclusive == EXCL && (dvp->v_flag & V_XATTRDIR)) {
6538                 return (EINVAL);
6539         }

6541         /* . and .. have special meaning in the protocol, reject them. */

6543         if (nm[0] == '.' && (nm[1] == '\0' || (nm[1] == '.' && nm[2] == '\0')))
6544                 return (EISDIR);

6546         drp = VTOR4(dvp);

6548         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp)))
6549                 return (EINTR);

6551 top:
6552         /*
6553          * We make a copy of the attributes because the caller does not
6554          * expect us to change what va points to.
6555          */
6556         vattr = *va;

6558         /*
6559          * If the pathname is "", then dvp is the root vnode of
6560          * a remote file mounted over a local directory.
6561          * All that needs to be done is access
6562          * checking and truncation.  Note that we avoid doing
6563          * open w/ create because the parent directory might
6564          * be in pseudo-fs and the open would fail.
6565          */
6566         if (*nm == '\0') {
6567                 error = 0;
6568                 VN_HOLD(dvp);
6569                 vp = dvp;
6570                 must_trunc = TRUE;
6571         } else {
6572                 /*
6573                  * We need to go over the wire, just to be sure whether the
6574                  * file exists or not.  Using the DNLC can be dangerous in
6575                  * this case when making a decision regarding existence.
6576                  */
6577                 error = nfs4lookup(dvp, nm, &vp, cr, 1);
6578         }

6580         if (exclusive)
6581                 createmode = EXCLUSIVE4;
6582         else
6583                 createmode = GUARDED4;

6585         /*
6586          * error would be set if the file does not exist on the
6587          * server, so lets go create it.
6588          */
6589         if (error) {
6590                 goto create_otw;
6591         }
```

```
6593         /*
6594          * File does exist on the server
6595          */
6596         if (exclusive == EXCL)
6597                 error = EEXIST;
6598         else if (vp->v_type == VDIR && (mode & VWRITE))
6599                 error = EISDIR;
6600         else {
6601                 /*
6602                  * If vnode is a device, create special vnode.
6603                  */
6604                 if (ISVDEV(vp->v_type)) {
6605                         tempvp = vp;
6606                         vp = specvp(vp, vp->v_rdev, vp->v_type, cr);
6607                         VN_RELE(tempvp);
6608                 }
6609                 if (!(error = VOP_ACCESS(vp, mode, 0, cr, ct))) {
6610                         if ((vattr.va_mask & AT_SIZE) &&
6611                             vp->v_type == VREG) {
6612                                 rp = VTOR4(vp);
6613                                 /*
6614                                  * Check here for large file handled
6615                                  * by LF-unaware process (as
6616                                  * ufs_create() does)
6617                                  */
6618                                 if (!(flags & FOFFMAX)) {
6619                                         mutex_enter(&rp->r_statelock);
6620                                         if (rp->r_size > MAXOFF32_T)
6621                                                 error = EOVERFLOW;
6622                                         mutex_exit(&rp->r_statelock);
6623                                 }

6625                                 /* if error is set then we need to return */
6626                                 if (error) {
6627                                         nfs_rw_exit(&drp->r_rwlock);
6628                                         VN_RELE(vp);
6629                                         return (error);
6630                                 }

6632                                 if (must_trunc) {
6633                                         vattr.va_mask = AT_SIZE;
6634                                         error = nfs4setattr(vp, &vattr, 0, cr,
6635                                             NULL);
6636                                 } else {
6637                                 /*
6638                                  * we know we have a regular file that already
6639                                  * exists and we may end up truncating the file
6640                                  * as a result of the open_otw, so flush out
6641                                  * any dirty pages for this file first.
6642                                  */
6643                                         if (nfs4_has_pages(vp) &&
6644                                             ((rp->r_flags & R4DIRTY) ||
6645                                             rp->r_count > 0 ||
6646                                             rp->r_mapcnt > 0)) {
6647                                                 error = nfs4_putpage(vp,
6648                                                     (offset_t)0, 0, 0, cr, ct);
6649                                                 if (error && (error == ENOSPC ||
6650                                                     error == EDQUOT)) {
6651                                                         mutex_enter(
6652                                                             &rp->r_statelock);
6653                                                         if (!rp->r_error)
6654                                                                 rp->r_error =
6655                                                                     error;
6656                                                         mutex_exit(
6657                                                             &rp->r_statelock);
6658                                                 }
```



```
6659                                         }
6660                                         vattr.va_mask = (AT_SIZE |
6661                                             AT_TYPE | AT_MODE);
6662                                         vattr.va_type = VREG;
6663                                         createmode = UNCHECKED4;
6664                                         truncating = 1;
6665                                         goto create_otw;
6666                                 }
6667                         }
6668                 }
6669         }
6670         nfs_rw_exit(&drp->r_rwlock);
6671         if (error) {
6672                 VN_RELE(vp);
6673         } else {
6674                 vnode_t *tvp;
6675                 rnode4_t *trp;
6676                 tvp = vp;
6677                 if (vp->v_type == VREG) {
6678                         trp = VTOR4(vp);
6679                         if (IS_SHADOW(vp, trp))
6680                                 tvp = RTOV4(trp);
6681                 }

6683                 if (must_trunc) {
6684                         /*
6685                          * existing file got truncated, notify.
6686                          */
6687                         vnevent_create(tvp, ct);
6688                 }

6690                 *vpp = vp;
6691         }
6692         return (error);

6694 create_otw:
6695         dnlc_remove(dvp, nm);

6697         ASSERT(vattr.va_mask & AT_TYPE);

6699         /*
6700          * If not a regular file let nfs4mknod() handle it.
6701          */
6702         if (vattr.va_type != VREG) {
6703                 error = nfs4mknod(dvp, nm, &vattr, exclusive, mode, vpp, cr);
6704                 nfs_rw_exit(&drp->r_rwlock);
6705                 return (error);
6706         }

6708         /*
6709          * It _is_ a regular file.
6710          */
6711         ASSERT(vattr.va_mask & AT_MODE);
6712         if (MANDMODE(vattr.va_mode)) {
6713                 nfs_rw_exit(&drp->r_rwlock);
6714                 return (EACCES);
6715         }

6717         /*
6718          * If this happens to be a mknod of a regular file, then flags will
6719          * have neither FREAD or FWRITE.  However, we must set at least one
6720          * for the call to nfs4open_otw.  If it's open(O_CREAT) driving
6721          * nfs4_create, then either FREAD, FWRITE, or FRDWR has already been
6722          * set (based on openmode specified by app).
6723          */
6724         if ((flags & (FREAD|FWRITE)) == 0)
```

```
6725                flags |= (FREAD|FWRITE);

6727        error = nfs4open_otw(dvp, nm, &vattr, vpp, cr, 1, flags, createmode, 0);

6729        if (vp != NULL) {
6730                /* if create was successful, throw away the file's pages */
6731                if (!error && (vattr.va_mask & AT_SIZE))
6732                        nfs4_invalidate_pages(vp, (vattr.va_size & PAGEMASK),
6733                            cr);
6734                /* release the lookup hold */
6735                VN_RELE(vp);
6736                vp = NULL;
6737        }

6739        /*
6740         * validate that we opened a regular file. This handles a misbehaving
6741         * server that returns an incorrect FH.
6742         */
6743        if ((error == 0) && *vpp && (*vpp)->v_type != VREG) {
6744                error = EISDIR;
6745                VN_RELE(*vpp);
6746        }

6748        /*
6749         * If this is not an exclusive create, then the CREATE
6750         * request will be made with the GUARDED mode set.  This
6751         * means that the server will return EEXIST if the file
6752         * exists.  The file could exist because of a retransmitted
6753         * request.  In this case, we recover by starting over and
6754         * checking to see whether the file exists.  This second
6755         * time through it should and a CREATE request will not be
6756         * sent.
6757         *
6758         * This handles the problem of a dangling CREATE request
6759         * which contains attributes which indicate that the file
6760         * should be truncated.  This retransmitted request could
6761         * possibly truncate valid data in the file if not caught
6762         * by the duplicate request mechanism on the server or if
6763         * not caught by other means.  The scenario is:
6764         *
6765         * Client transmits CREATE request with size = 0
6766         * Client times out, retransmits request.
6767         * Response to the first request arrives from the server
6768         *  and the client proceeds on.
6769         * Client writes data to the file.
6770         * The server now processes retransmitted CREATE request
6771         *  and truncates file.
6772         *
6773         * The use of the GUARDED CREATE request prevents this from
6774         * happening because the retransmitted CREATE would fail
6775         * with EEXIST and would not truncate the file.
6776         */
6777        if (error == EEXIST && exclusive == NONEXCL) {
6778 #ifdef DEBUG
6779                nfs4_create_misses++;
6780 #endif
6781                goto top;
6782        }
6783        nfs_rw_exit(&drp->r_rwlock);
6784        if (truncating && !error && *vpp) {
6785                vnode_t *tvp;
6786                rnode4_t *trp;
6787                /*
6788                 * existing file got truncated, notify.
6789                 */
6790                tvp = *vpp;
```



```
6791                trp = VTOR4(tvp);
6792                if (IS_SHADOW(tvp, trp))
6793                        tvp = RTOV4(trp);
6794                vnevent_create(tvp, ct);
6795        }
6796        return (error);
6797 }

6799 /*
6800  * Create compound (for mkdir, mknod, symlink):
6801  * { Putfh <dfh>; Create; Getfh; Getattr }
6802  * It's okay if setattr failed to set gid - this is not considered
6803  * an error, but purge attrs in that case.
6804  */
6805 static int
6806 call_nfs4_create_req(vnode_t *dvp, char *nm, void *data, struct vattr *va,
6807     vnode_t **vpp, cred_t *cr, nfs_ftype4 type)
6808 {
6809        int need_end_op = FALSE;
6810        COMPOUND4args_clnt args;
6811        COMPOUND4res_clnt res, *resp = NULL;
6812        nfs_argop4 *argop;
6813        nfs_resop4 *resop;
6814        int doqueue;
6815        mntinfo4_t *mi;
6816        rnode4_t *drp = VTOR4(dvp);
6817        change_info4 *cinfo;
6818        GETFH4res *gf_res;
6819        struct vattr vattr;
6820        vnode_t *vp;
6821        fattr4 *crattr;
6822        bool_t needrecov = FALSE;
6823        nfs4_recov_state_t recov_state;
6824        nfs4_sharedfh_t *sfhp = NULL;
6825        hrtime_t t;
6826        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
6827        int numops, argoplist_size, setgid_flag, idx_create, idx_fattr;
6828        dirattr_info_t dinfo, *dinfop;
6829        servinfo4_t *svp;
6830        bitmap4 supp_attrs;

6832        ASSERT(type == NF4DIR || type == NF4LNK || type == NF4BLK ||
6833            type == NF4CHR || type == NF4SOCK || type == NF4FIFO);

6835        mi = VTOMI4(dvp);

6837        /*
6838         * Make sure we properly deal with setting the right gid
6839         * on a new directory to reflect the parent's setgid bit
6840         */
6841        setgid_flag = 0;
6842        if (type == NF4DIR) {
6843                struct vattr dva;

6845                va->va_mode &= ~VSGID;
6846                dva.va_mask = AT_MODE | AT_GID;
6847                if (VOP_GETATTR(dvp, &dva, 0, cr, NULL) == 0) {

6849                        /*
6850                         * If the parent's directory has the setgid bit set
6851                         * _and_ the client was able to get a valid mapping
6852                         * for the parent dir's owner_group, we want to
6853                         * append NVERIFY(owner_group == dva.va_gid) and
6854                         * SETTATTR to the CREATE compound.
6855                         */
6856                        if (mi->mi_flags & MI4_GRPID || dva.va_mode & VSGID) {
```

```
6857                                setgid_flag = 1;
6858                                va->va_mode |= VSGID;
6859                                if (dva.va_gid != GID_NOBODY) {
6860                                        va->va_mask |= AT_GID;
6861                                        va->va_gid = dva.va_gid;
6862                                }
6863                        }
6864                }
6865        }

6867        /*
6868         * Create ops:
6869         *      0:putfh(dir) 1:savefh(dir) 2:create 3:getfh(new) 4:getattr(new)
6870         *      5:restorefh(dir) 6:getattr(dir)
6871         *
6872         * if (setgid)
6873         *      0:putfh(dir) 1:create 2:getfh(new) 3:getattr(new)
6874         *      4:savefh(new) 5:putfh(dir) 6:getattr(dir) 7:restorefh(new)
6875         *      8:nverify 9:setattr
6876         */
6877        if (setgid_flag) {
6878                numops = 10;
6879                idx_create = 1;
6880                idx_fattr = 3;
6881        } else {
6882                numops = 7;
6883                idx_create = 2;
6884                idx_fattr = 4;
6885        }

6887        ASSERT(nfs_zone() == mi->mi_zone);
6888        if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp))) {
6889                return (EINTR);
6890        }
6891        recov_state.rs_flags = 0;
6892        recov_state.rs_num_retry_despite_err = 0;

6894        argoplist_size = numops * sizeof (nfs_argop4);
6895        argop = kmem_alloc(argoplist_size, KM_SLEEP);

6897 recov_retry:
6898        if (type == NF4LNK)
6899                args.ctag = TAG_SYMLINK;
6900        else if (type == NF4DIR)
6901                args.ctag = TAG_MKDIR;
6902        else
6903                args.ctag = TAG_MKNOD;

6905        args.array_len = numops;
6906        args.array = argop;

6908        if (e.error = nfs4_start_op(mi, dvp, NULL, &recov_state)) {
6909                nfs_rw_exit(&drp->r_rwlock);
6910                kmem_free(argop, argoplist_size);
6911                return (e.error);
6912        }
6913        need_end_op = TRUE;


6916        /* 0: putfh directory */
6917        argop[0].argop = OP_CPUTFH;
6918        argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

6920        /* 1/2: Create object */
6921        argop[idx_create].argop = OP_CCREATE;
6922        argop[idx_create].nfs_argop4_u.opccreate.cname = nm;
```



```
6923        argop[idx_create].nfs_argop4_u.opccreate.type = type;
6924        if (type == NF4LNK) {
6925                /*
6926                 * symlink, treat name as data
6927                 */
6928                ASSERT(data != NULL);
6929                argop[idx_create].nfs_argop4_u.opccreate.ftype4_u.clinkdata =
6930                    (char *)data;
6931        }
6932        if (type == NF4BLK || type == NF4CHR) {
6933                ASSERT(data != NULL);
6934                argop[idx_create].nfs_argop4_u.opccreate.ftype4_u.devdata =
6935                    *((specdata4 *)data);
6936        }

6938        crattr = &argop[idx_create].nfs_argop4_u.opccreate.createattrs;

6940        svp = drp->r_server;
6941        (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
6942        supp_attrs = svp->sv_supp_attrs;
6943        nfs_rw_exit(&svp->sv_lock);

6945        if (vattr_to_fattr4(va, NULL, crattr, 0, OP_CREATE, supp_attrs)) {
6946                nfs_rw_exit(&drp->r_rwlock);
6947                nfs4_end_op(mi, dvp, NULL, &recov_state, needrecov);
6948                e.error = EINVAL;
6949                kmem_free(argop, argoplist_size);
6950                return (e.error);
6951        }

6953        /* 2/3: getfh fh of created object */
6954        ASSERT(idx_create + 1 == idx_fattr - 1);
6955        argop[idx_create + 1].argop = OP_GETFH;

6957        /* 3/4: getattr of new object */
6958        argop[idx_fattr].argop = OP_GETATTR;
6959        argop[idx_fattr].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
6960        argop[idx_fattr].nfs_argop4_u.opgetattr.mi = mi;

6962        if (setgid_flag) {
6963                vattr_t _v;

6965                argop[4].argop = OP_SAVEFH;

6967                argop[5].argop = OP_CPUTFH;
6968                argop[5].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

6970                argop[6].argop = OP_GETATTR;
6971                argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
6972                argop[6].nfs_argop4_u.opgetattr.mi = mi;

6974                argop[7].argop = OP_RESTOREFH;

6976                /*
6977                 * nverify
6978                 *
6979                 * XXX - Revisit the last argument to nfs4_end_op()
6980                 *       once 5020486 is fixed.
6981                 */
6982                _v.va_mask = AT_GID;
6983                _v.va_gid = va->va_gid;
6984                if (e.error = nfs4args_verify(&argop[8], &_v, OP_NVERIFY,
6985                    supp_attrs)) {
6986                        nfs4_end_op(mi, dvp, *vpp, &recov_state, TRUE);
6987                        nfs_rw_exit(&drp->r_rwlock);
6988                        nfs4_fattr4_free(crattr);
```

```
6989                        kmem_free(argop, argoplist_size);
6990                        return (e.error);
6991                }

6993                /*
6994                 * setattr
6995                 *
6996                 * We _know_ we're not messing with AT_SIZE or AT_XTIME,
6997                 * so no need for stateid or flags. Also we specify NULL
6998                 * rp since we're only interested in setting owner_group
6999                 * attributes.
7000                 */
7001                nfs4args_setattr(&argop[9], &_v, NULL, 0, NULL, cr, supp_attrs,
7002                    &e.error, 0);

7004                if (e.error) {
7005                        nfs4_end_op(mi, dvp, *vpp, &recov_state, TRUE);
7006                        nfs_rw_exit(&drp->r_rwlock);
7007                        nfs4_fattr4_free(crattr);
7008                        nfs4args_verify_free(&argop[8]);
7009                        kmem_free(argop, argoplist_size);
7010                        return (e.error);
7011                }
7012        } else {
7013                argop[1].argop = OP_SAVEFH;

7015                argop[5].argop = OP_RESTOREFH;

7017                argop[6].argop = OP_GETATTR;
7018                argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7019                argop[6].nfs_argop4_u.opgetattr.mi = mi;
7020        }

7022        dnlc_remove(dvp, nm);

7024        doqueue = 1;
7025        t = gethrtime();
7026        rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

7028        needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
7029        if (e.error) {
7030                PURGE_ATTRCACHE4(dvp);
7031                if (!needrecov)
7032                        goto out;
7033        }

7035        if (needrecov) {
7036                if (nfs4_start_recovery(&e, mi, dvp, NULL, NULL, NULL,
7037                    OP_CREATE, NULL, NULL, NULL) == FALSE) {
7038                        nfs4_end_op(mi, dvp, NULL, &recov_state,
7039                            needrecov);
7040                        need_end_op = FALSE;
7041                        nfs4_fattr4_free(crattr);
7042                        if (setgid_flag) {
7043                                nfs4args_verify_free(&argop[8]);
7044                                nfs4args_setattr_free(&argop[9]);
7045                        }
7046                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
7047                        goto recov_retry;
7048                }
7049        }

7051        resp = &res;

7053        if (res.status != NFS4_OK && res.array_len <= idx_fattr + 1) {
```



```
7055                if (res.status == NFS4ERR_BADOWNER)
7056                        nfs4_log_badowner(mi, OP_CREATE);

7058                e.error = geterrno4(res.status);

7060                /*
7061                 * This check is left over from when create was implemented
7062                 * using a setattr op (instead of createattrs).  If the
7063                 * putfh/create/getfh failed, the error was returned.  If
7064                 * setattr/getattr failed, we keep going.
7065                 *
7066                 * It might be better to get rid of the GETFH also, and just
7067                 * do PUTFH/CREATE/GETATTR since the FH attr is mandatory.
7068                 * Then if any of the operations failed, we could return the
7069                 * error now, and remove much of the error code below.
7070                 */
7071                if (res.array_len <= idx_fattr) {
7072                        /*
7073                         * Either Putfh, Create or Getfh failed.
7074                         */
7075                        PURGE_ATTRCACHE4(dvp);
7076                        /*
7077                         * nfs4_purge_stale_fh() may generate otw calls through
7078                         * nfs4_invalidate_pages. Hence the need to call
7079                         * nfs4_end_op() here to avoid nfs4_start_op() deadlock.
7080                         */
7081                        nfs4_end_op(mi, dvp, NULL, &recov_state,
7082                            needrecov);
7083                        need_end_op = FALSE;
7084                        nfs4_purge_stale_fh(e.error, dvp, cr);
7085                        goto out;
7086                }
7087        }

7089        resop = &res.array[idx_create]; /* create res */
7090        cinfo = &resop->nfs_resop4_u.opcreate.cinfo;

7092        resop = &res.array[idx_create + 1]; /* getfh res */
7093        gf_res = &resop->nfs_resop4_u.opgetfh;

7095        sfhp = sfh4_get(&gf_res->object, mi);
7096        if (e.error) {
7097                *vpp = vp = makenfs4node(sfhp, NULL, dvp->v_vfsp, t, cr, dvp,
7098                    fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
7099                if (vp->v_type == VNON) {
7100                        vattr.va_mask = AT_TYPE;
7101                        /*
7102                         * Need to call nfs4_end_op before nfs4getattr to avoid
7103                         * potential nfs4_start_op deadlock. See RFE 4777612.
7104                         */
7105                        nfs4_end_op(mi, dvp, NULL, &recov_state,
7106                            needrecov);
7107                        need_end_op = FALSE;
7108                        e.error = nfs4getattr(vp, &vattr, cr);
7109                        if (e.error) {
7110                                VN_RELE(vp);
7111                                *vpp = NULL;
7112                                goto out;
7113                        }
7114                        vp->v_type = vattr.va_type;
7115                }
7116                e.error = 0;
7117        } else {
7118                *vpp = vp = makenfs4node(sfhp,
7119                    &res.array[idx_fattr].nfs_resop4_u.opgetattr.ga_res,
7120                    dvp->v_vfsp, t, cr,
```

```
7121                    dvp, fn_get(VTOSV(dvp)->sv_name, nm, sfhp));
7122        }

7124        /*
7125         * If compound succeeded, then update dir attrs
7126         */
7127        if (res.status == NFS4_OK) {
7128                dinfo.di_garp = &res.array[6].nfs_resop4_u.opgetattr.ga_res;
7129                dinfo.di_cred = cr;
7130                dinfo.di_time_call = t;
7131                dinfop = &dinfo;
7132        } else
7133                dinfop = NULL;

7135        /* Update directory cache attribute, readdir and dnlc caches */
7136        nfs4_update_dircaches(cinfo, dvp, vp, nm, dinfop);

7138 out:
7139        if (sfhp != NULL)
7140                sfh4_rele(&sfhp);
7141        nfs_rw_exit(&drp->r_rwlock);
7142        nfs4_fattr4_free(crattr);
7143        if (setgid_flag) {
7144                nfs4args_verify_free(&argop[8]);
7145                nfs4args_setattr_free(&argop[9]);
7146        }
7147        if (resp)
7148                (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
7149        if (need_end_op)
7150                nfs4_end_op(mi, dvp, NULL, &recov_state, needrecov);

7152        kmem_free(argop, argoplist_size);
7153        return (e.error);
7154 }

7156 /* ARGSUSED */
7157 static int
7158 nfs4mknod(vnode_t *dvp, char *nm, struct vattr *va, enum vcexcl exclusive,
7159     int mode, vnode_t **vpp, cred_t *cr)
7160 {
7161        int error;
7162        vnode_t *vp;
7163        nfs_ftype4 type;
7164        specdata4 spec, *specp = NULL;

7166        ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);

7168        switch (va->va_type) {
7169        case VCHR:
7170        case VBLK:
7171                type = (va->va_type == VCHR) ? NF4CHR : NF4BLK;
7172                spec.specdata1 = getmajor(va->va_rdev);
7173                spec.specdata2 = getminor(va->va_rdev);
7174                specp = &spec;
7175                break;

7177        case VFIFO:
7178                type = NF4FIFO;
7179                break;
7180        case VSOCK:
7181                type = NF4SOCK;
7182                break;

7184        default:
7185                return (EINVAL);
7186        }
```



```
7188        error = call_nfs4_create_req(dvp, nm, specp, va, &vp, cr, type);
7189        if (error) {
7190                return (error);
7191        }

7193        /*
7194         * This might not be needed any more; special case to deal
7195         * with problematic v2/v3 servers.  Since create was unable
7196         * to set group correctly, not sure what hope setattr has.
7197         */
7198        if (va->va_gid != VTOR4(vp)->r_attr.va_gid) {
7199                va->va_mask = AT_GID;
7200                (void) nfs4setattr(vp, va, 0, cr, NULL);
7201        }

7203        /*
7204         * If vnode is a device create special vnode
7205         */
7206        if (ISVDEV(vp->v_type)) {
7207                *vpp = specvp(vp, vp->v_rdev, vp->v_type, cr);
7208                VN_RELE(vp);
7209        } else {
7210                *vpp = vp;
7211        }
7212        return (error);
7213 }

7215 /*
7216  * Remove requires that the current fh be the target directory.
7217  * After the operation, the current fh is unchanged.
7218  * The compound op structure is:
7219  *      PUTFH(targetdir), REMOVE
7220  *
7221  * Weirdness: if the vnode to be removed is open
7222  * we rename it instead of removing it and nfs_inactive
7223  * will remove the new name.
7224  */
7225 /* ARGSUSED */
7226 static int
7227 nfs4_remove(vnode_t *dvp, char *nm, cred_t *cr, caller_context_t *ct, int flags)
7228 {
7229        COMPOUND4args_clnt args;
7230        COMPOUND4res_clnt res, *resp = NULL;
7231        REMOVE4res *rm_res;
7232        nfs_argop4 argop[3];
7233        nfs_resop4 *resop;
7234        vnode_t *vp;
7235        char *tmpname;
7236        int doqueue;
7237        mntinfo4_t *mi;
7238        rnode4_t *rp;
7239        rnode4_t *drp;
7240        int needrecov = 0;
7241        nfs4_recov_state_t recov_state;
7242        int isopen;
7243        nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
7244        dirattr_info_t dinfo;

7246        if (nfs_zone() != VTOMI4(dvp)->mi_zone)
7247                return (EPERM);
7248        drp = VTOR4(dvp);
7249        if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp)))
7250                return (EINTR);

7252        e.error = nfs4lookup(dvp, nm, &vp, cr, 0);
```

```
7253         if (e.error) {
7254                 nfs_rw_exit(&drp->r_rwlock);
7255                 return (e.error);
7256         }
7257
7258         if (vp->v_type == VDIR) {
7259                 VN_RELE(vp);
7260                 nfs_rw_exit(&drp->r_rwlock);
7261                 return (EISDIR);
7262         }
7263
7264         /*
7265          * First just remove the entry from the name cache, as it
7266          * is most likely the only entry for this vp.
7267          */
7268         dnlc_remove(dvp, nm);
7269
7270         rp = VTOR4(vp);
7271
7272         /*
7273          * For regular file types, check to see if the file is open by looking
7274          * at the open streams.
7275          * For all other types, check the reference count on the vnode.  Since
7276          * they are not opened OTW they never have an open stream.
7277          *
7278          * If the file is open, rename it to .nfsXXXX.
7279          */
7280         if (vp->v_type != VREG) {
7281                 /*
7282                  * If the file has a v_count > 1 then there may be more than one
7283                  * entry in the name cache due multiple links or an open file,
7284                  * but we don't have the real reference count so flush all
7285                  * possible entries.
7286                  */
7287                 if (vp->v_count > 1)
7288                         dnlc_purge_vp(vp);
7289
7290                 /*
7291                  * Now we have the real reference count.
7292                  */
7293                 isopen = vp->v_count > 1;
7294         } else {
7295                 mutex_enter(&rp->r_os_lock);
7296                 isopen = list_head(&rp->r_open_streams) != NULL;
7297                 mutex_exit(&rp->r_os_lock);
7298         }
7299
7300         mutex_enter(&rp->r_statelock);
7301         if (isopen &&
7302             (rp->r_unldvp == NULL || strcmp(nm, rp->r_unlname) == 0)) {
7303                 mutex_exit(&rp->r_statelock);
7304                 tmpname = newname();
7305                 e.error = nfs4rename(dvp, nm, dvp, tmpname, cr, ct);
7306                 if (e.error)
7307                         kmem_free(tmpname, MAXNAMELEN);
7308                 else {
7309                         mutex_enter(&rp->r_statelock);
7310                         if (rp->r_unldvp == NULL) {
7311                                 VN_HOLD(dvp);
7312                                 rp->r_unldvp = dvp;
7313                                 if (rp->r_unlcred != NULL)
7314                                         crfree(rp->r_unlcred);
7315                                 crhold(cr);
7316                                 rp->r_unlcred = cr;
7317                                 rp->r_unlname = tmpname;
7318                         } else {
```



```
7319                                 kmem_free(rp->r_unlname, MAXNAMELEN);
7320                                 rp->r_unlname = tmpname;
7321                         }
7322                         mutex_exit(&rp->r_statelock);
7323                 }
7324                 VN_RELE(vp);
7325                 nfs_rw_exit(&drp->r_rwlock);
7326                 return (e.error);
7327         }
7328         /*
7329          * Actually remove the file/dir
7330          */
7331         mutex_exit(&rp->r_statelock);
7332
7333         /*
7334          * We need to flush any dirty pages which happen to
7335          * be hanging around before removing the file.
7336          * This shouldn't happen very often since in NFSv4
7337          * we should be close to open consistent.
7338          */
7339         if (nfs4_has_pages(vp) &&
7340             ((rp->r_flags & R4DIRTY) || rp->r_count > 0)) {
7341                 e.error = nfs4_putpage(vp, (u_offset_t)0, 0, 0, cr, ct);
7342                 if (e.error && (e.error == ENOSPC || e.error == EDQUOT)) {
7343                         mutex_enter(&rp->r_statelock);
7344                         if (!rp->r_error)
7345                                 rp->r_error = e.error;
7346                         mutex_exit(&rp->r_statelock);
7347                 }
7348         }
7349
7350         mi = VTOMI4(dvp);
7351
7352         (void) nfs4delegreturn(rp, NFS4_DR_REOPEN);
7353         recov_state.rs_flags = 0;
7354         recov_state.rs_num_retry_despite_err = 0;
7355
7356 recov_retry:
7357         /*
7358          * Remove ops: putfh dir; remove
7359          */
7360         args.ctag = TAG_REMOVE;
7361         args.array_len = 3;
7362         args.array = argop;
7363
7364         e.error = nfs4_start_op(VTOMI4(dvp), dvp, NULL, &recov_state);
7365         if (e.error) {
7366                 nfs_rw_exit(&drp->r_rwlock);
7367                 VN_RELE(vp);
7368                 return (e.error);
7369         }
7370
7371         /* putfh directory */
7372         argop[0].argop = OP_CPUTFH;
7373         argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;
7374
7375         /* remove */
7376         argop[1].argop = OP_CREMOVE;
7377         argop[1].nfs_argop4_u.opcremove.ctarget = nm;
7378
7379         /* getattr dir */
7380         argop[2].argop = OP_GETATTR;
7381         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7382         argop[2].nfs_argop4_u.opgetattr.mi = mi;
7383
7384         doqueue = 1;
```

```
7385         dinfo.di_time_call = gethrtime();
7386         rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

7388         PURGE_ATTRCACHE4(vp);

7390         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
7391         if (e.error)
7392                 PURGE_ATTRCACHE4(dvp);

7394         if (needrecov) {
7395                 if (nfs4_start_recovery(&e, VTOMI4(dvp), dvp,
7396                     NULL, NULL, NULL, OP_REMOVE, NULL, NULL, NULL) == FALSE) {
7397                         if (!e.error)
7398                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
7399                                     (caddr_t)&res);
7400                         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state,
7401                             needrecov);
7402                         goto recov_retry;
7403                 }
7404         }

7406         /*
7407          * Matching nfs4_end_op() for start_op() above.
7408          * There is a path in the code below which calls
7409          * nfs4_purge_stale_fh(), which may generate otw calls through
7410          * nfs4_invalidate_pages. Hence we need to call nfs4_end_op()
7411          * here to avoid nfs4_start_op() deadlock.
7412          */
7413         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);

7415         if (!e.error) {
7416                 resp = &res;

7418                 if (res.status) {
7419                         e.error = geterrno4(res.status);
7420                         PURGE_ATTRCACHE4(dvp);
7421                         nfs4_purge_stale_fh(e.error, dvp, cr);
7422                 } else {
7423                         resop = &res.array[1];  /* remove res */
7424                         rm_res = &resop->nfs_resop4_u.opremove;

7426                         dinfo.di_garp =
7427                             &res.array[2].nfs_resop4_u.opgetattr.ga_res;
7428                         dinfo.di_cred = cr;

7430                         /* Update directory attr, readdir and dnlc caches */
7431                         nfs4_update_dircaches(&rm_res->cinfo, dvp, NULL, NULL,
7432                             &dinfo);
7433                 }
7434         }
7435         nfs_rw_exit(&drp->r_rwlock);
7436         if (resp)
7437                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);

7439         if (e.error == 0) {
7440                 vnode_t *tvp;
7441                 rnode4_t *trp;
7442                 trp = VTOR4(vp);
7443                 tvp = vp;
7444                 if (IS_SHADOW(vp, trp))
7445                         tvp = RTOV4(trp);
7446                 vnevent_remove(tvp, dvp, nm, ct);
7447         }
7448         VN_RELE(vp);
7449         return (e.error);
7450 }
```



```
7452 /*
7453  * Link requires that the current fh be the target directory and the
7454  * saved fh be the source fh. After the operation, the current fh is unchanged.
7455  * Thus the compound op structure is:
7456  *      PUTFH(file), SAVEFH, PUTFH(targetdir), LINK, RESTOREFH,
7457  *      GETATTR(file)
7458  */
7459 /* ARGSUSED */
7460 static int
7461 nfs4_link(vnode_t *tdvp, vnode_t *svp, char *tnm, cred_t *cr,
7462     caller_context_t *ct, int flags)
7463 {
7464         COMPOUND4args_clnt args;
7465         COMPOUND4res_clnt res, *resp = NULL;
7466         LINK4res *ln_res;
7467         int argoplist_size  = 7 * sizeof (nfs_argop4);
7468         nfs_argop4 *argop;
7469         nfs_resop4 *resop;
7470         vnode_t *realvp, *nvp;
7471         int doqueue;
7472         mntinfo4_t *mi;
7473         rnode4_t *tdrp;
7474         bool_t needrecov = FALSE;
7475         nfs4_recov_state_t recov_state;
7476         hrtime_t t;
7477         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
7478         dirattr_info_t dinfo;

7480         ASSERT(*tnm != '\0');
7481         ASSERT(tdvp->v_type == VDIR);
7482         ASSERT(nfs4_consistent_type(tdvp));
7483         ASSERT(nfs4_consistent_type(svp));

7485         if (nfs_zone() != VTOMI4(tdvp)->mi_zone)
7486                 return (EPERM);
7487         if (VOP_REALVP(svp, &realvp, ct) == 0) {
7488                 svp = realvp;
7489                 ASSERT(nfs4_consistent_type(svp));
7490         }

7492         tdrp = VTOR4(tdvp);
7493         mi = VTOMI4(svp);

7495         if (!(mi->mi_flags & MI4_LINK)) {
7496                 return (EOPNOTSUPP);
7497         }
7498         recov_state.rs_flags = 0;
7499         recov_state.rs_num_retry_despite_err = 0;

7501         if (nfs_rw_enter_sig(&tdrp->r_rwlock, RW_WRITER, INTR4(tdvp)))
7502                 return (EINTR);

7504 recov_retry:
7505         argop = kmem_alloc(argoplist_size, KM_SLEEP);

7507         args.ctag = TAG_LINK;

7509         /*
7510          * Link ops: putfh fl; savefh; putfh tdir; link; getattr(dir);
7511          * restorefh; getattr(fl)
7512          */
7513         args.array_len = 7;
7514         args.array = argop;

7516         e.error = nfs4_start_op(VTOMI4(svp), svp, tdvp, &recov_state);
```

```
7517         if (e.error) {
7518                 kmem_free(argop, argoplist_size);
7519                 nfs_rw_exit(&tdrp->r_rwlock);
7520                 return (e.error);
7521         }

7523         /* 0. putfh file */
7524         argop[0].argop = OP_CPUTFH;
7525         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(svp)->r_fh;

7527         /* 1. save current fh to free up the space for the dir */
7528         argop[1].argop = OP_SAVEFH;

7530         /* 2. putfh targetdir */
7531         argop[2].argop = OP_CPUTFH;
7532         argop[2].nfs_argop4_u.opcputfh.sfh = tdrp->r_fh;

7534         /* 3. link: current_fh is targetdir, saved_fh is source */
7535         argop[3].argop = OP_CLINK;
7536         argop[3].nfs_argop4_u.opclink.cnewname = tnm;

7538         /* 4. Get attributes of dir */
7539         argop[4].argop = OP_GETATTR;
7540         argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7541         argop[4].nfs_argop4_u.opgetattr.mi = mi;

7543         /* 5. If link was successful, restore current vp to file */
7544         argop[5].argop = OP_RESTOREFH;

7546         /* 6. Get attributes of linked object */
7547         argop[6].argop = OP_GETATTR;
7548         argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
7549         argop[6].nfs_argop4_u.opgetattr.mi = mi;

7551         dnlc_remove(tdvp, tnm);

7553         doqueue = 1;
7554         t = gethrtime();

7556         rfs4call(VTOMI4(svp), &args, &res, cr, &doqueue, 0, &e);

7558         needrecov = nfs4_needs_recovery(&e, FALSE, svp->v_vfsp);
7559         if (e.error != 0 && !needrecov) {
7560                 PURGE_ATTRCACHE4(tdvp);
7561                 PURGE_ATTRCACHE4(svp);
7562                 nfs4_end_op(VTOMI4(svp), svp, tdvp, &recov_state, needrecov);
7563                 goto out;
7564         }

7566         if (needrecov) {
7567                 bool_t abort;

7569                 abort = nfs4_start_recovery(&e, VTOMI4(svp), svp, tdvp,
7570                     NULL, NULL, OP_LINK, NULL, NULL, NULL);
7571                 if (abort == FALSE) {
7572                         nfs4_end_op(VTOMI4(svp), svp, tdvp, &recov_state,
7573                             needrecov);
7574                         kmem_free(argop, argoplist_size);
7575                         if (!e.error)
7576                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
7577                                     (caddr_t)&res);
7578                         goto recov_retry;
7579                 } else {
7580                         if (e.error != 0) {
7581                                 PURGE_ATTRCACHE4(tdvp);
7582                                 PURGE_ATTRCACHE4(svp);
```



```
7583                                 nfs4_end_op(VTOMI4(svp), svp, tdvp,
7584                                     &recov_state, needrecov);
7585                                 goto out;
7586                         }
7587                         /* fall through for res.status case */
7588                 }
7589         }

7591         nfs4_end_op(VTOMI4(svp), svp, tdvp, &recov_state, needrecov);

7593         resp = &res;
7594         if (res.status) {
7595                 /* If link succeeded, then don't return error */
7596                 e.error = geterrno4(res.status);
7597                 if (res.array_len <= 4) {
7598                         /*
7599                          * Either Putfh, Savefh, Putfh dir, or Link failed
7600                          */
7601                         PURGE_ATTRCACHE4(svp);
7602                         PURGE_ATTRCACHE4(tdvp);
7603                         if (e.error == EOPNOTSUPP) {
7604                                 mutex_enter(&mi->mi_lock);
7605                                 mi->mi_flags &= ~MI4_LINK;
7606                                 mutex_exit(&mi->mi_lock);
7607                         }
7608                         /* Remap EISDIR to EPERM for non-root user for SVVS */
7609                         /* XXX-LP */
7610                         if (e.error == EISDIR && crgetuid(cr) != 0)
7611                                 e.error = EPERM;
7612                         goto out;
7613                 }
7614         }

7616         /* either no error or one of the postop getattr failed */

7618         /*
7619          * XXX - if LINK succeeded, but no attrs were returned for link
7620          * file, purge its cache.
7621          *
7622          * XXX Perform a simplified version of wcc checking. Instead of
7623          * have another getattr to get pre-op, just purge cache if
7624          * any of the ops prior to and including the getattr failed.
7625          * If the getattr succeeded then update the attrcache accordingly.
7626          */

7628         /*
7629          * update cache with link file postattrs.
7630          * Note: at this point resop points to link res.
7631          */
7632         resop = &res.array[3];  /* link res */
7633         ln_res = &resop->nfs_resop4_u.oplink;
7634         if (res.status == NFS4_OK)
7635                 e.error = nfs4_update_attrcache(res.status,
7636                     &res.array[6].nfs_resop4_u.opgetattr.ga_res,
7637                     t, svp, cr);

7639         /*
7640          * Call makenfs4node to create the new shadow vp for tnm.
7641          * We pass NULL attrs because we just cached attrs for
7642          * the src object.  All we're trying to accomplish is to
7643          * to create the new shadow vnode.
7644          */
7645         nvp = makenfs4node(VTOR4(svp)->r_fh, NULL, tdvp->v_vfsp, t, cr,
7646             tdvp, fn_get(VTOSV(tdvp)->sv_name, tnm, VTOR4(svp)->r_fh));

7648         /* Update target cache attribute, readdir and dnlc caches */
```

```
7649         dinfo.di_garp = &res.array[4].nfs_resop4_u.opgetattr.ga_res;
7650         dinfo.di_time_call = t;
7651         dinfo.di_cred = cr;

7653         nfs4_update_dircaches(&ln_res->cinfo, tdvp, nvp, tnm, &dinfo);
7654         ASSERT(nfs4_consistent_type(tdvp));
7655         ASSERT(nfs4_consistent_type(svp));
7656         ASSERT(nfs4_consistent_type(nvp));
7657         VN_RELE(nvp);

7659         if (!e.error) {
7660                 vnode_t *tvp;
7661                 rnode4_t *trp;
7662                 /*
7663                  * Notify the source file of this link operation.
7664                  */
7665                 trp = VTOR4(svp);
7666                 tvp = svp;
7667                 if (IS_SHADOW(svp, trp))
7668                         tvp = RTOV4(trp);
7669                 vnevent_link(tvp, ct);
7670         }
7671 out:
7672         kmem_free(argop, argoplist_size);
7673         if (resp)
7674                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);

7676         nfs_rw_exit(&tdrp->r_rwlock);

7678         return (e.error);
7679 }

7681 /* ARGSUSED */
7682 static int
7683 nfs4_rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
7684     caller_context_t *ct, int flags)
7685 {
7686         vnode_t *realvp;

7688         if (nfs_zone() != VTOMI4(odvp)->mi_zone)
7689                 return (EPERM);
7690         if (VOP_REALVP(ndvp, &realvp, ct) == 0)
7691                 ndvp = realvp;

7693         return (nfs4rename(odvp, onm, ndvp, nnm, cr, ct));
7694 }

7696 /*
7697  * nfs4rename does the real work of renaming in NFS Version 4.
7698  *
7699  * A file handle is considered volatile for renaming purposes if either
7700  * of the volatile bits are turned on. However, the compound may differ
7701  * based on the likelihood of the filehandle to change during rename.
7702  */
7703 static int
7704 nfs4rename(vnode_t *odvp, char *onm, vnode_t *ndvp, char *nnm, cred_t *cr,
7705     caller_context_t *ct)
7706 {
7707         int error;
7708         mntinfo4_t *mi;
7709         vnode_t *nvp = NULL;
7710         vnode_t *ovp = NULL;
7711         char *tmpname = NULL;
7712         rnode4_t *rp;
7713         rnode4_t *odrp;
7714         rnode4_t *ndrp;
```



```
7715         int did_link = 0;
7716         int do_link = 1;
7717         nfsstat4 stat = NFS4_OK;

7719         ASSERT(nfs_zone() == VTOMI4(odvp)->mi_zone);
7720         ASSERT(nfs4_consistent_type(odvp));
7721         ASSERT(nfs4_consistent_type(ndvp));

7723         if (onm[0] == '.' && (onm[1] == '\0' ||
7724             (onm[1] == '.' && onm[2] == '\0')))
7725                 return (EINVAL);

7727         if (nnm[0] == '.' && (nnm[1] == '\0' ||
7728             (nnm[1] == '.' && nnm[2] == '\0')))
7729                 return (EINVAL);

7731         odrp = VTOR4(odvp);
7732         ndrp = VTOR4(ndvp);
7733         if ((intptr_t)odrp < (intptr_t)ndrp) {
7734                 if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR4(odvp)))
7735                         return (EINTR);
7736                 if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR4(ndvp))) {
7737                         nfs_rw_exit(&odrp->r_rwlock);
7738                         return (EINTR);
7739                 }
7740         } else {
7741                 if (nfs_rw_enter_sig(&ndrp->r_rwlock, RW_WRITER, INTR4(ndvp)))
7742                         return (EINTR);
7743                 if (nfs_rw_enter_sig(&odrp->r_rwlock, RW_WRITER, INTR4(odvp))) {
7744                         nfs_rw_exit(&ndrp->r_rwlock);
7745                         return (EINTR);
7746                 }
7747         }

7749         /*
7750          * Lookup the target file.  If it exists, it needs to be
7751          * checked to see whether it is a mount point and whether
7752          * it is active (open).
7753          */
7754         error = nfs4lookup(ndvp, nnm, &nvp, cr, 0);
7755         if (!error) {
7756                 int     isactive;

7758                 ASSERT(nfs4_consistent_type(nvp));
7759                 /*
7760                  * If this file has been mounted on, then just
7761                  * return busy because renaming to it would remove
7762                  * the mounted file system from the name space.
7763                  */
7764                 if (vn_ismntpt(nvp)) {
7765                         VN_RELE(nvp);
7766                         nfs_rw_exit(&odrp->r_rwlock);
7767                         nfs_rw_exit(&ndrp->r_rwlock);
7768                         return (EBUSY);
7769                 }

7771                 /*
7772                  * First just remove the entry from the name cache, as it
7773                  * is most likely the only entry for this vp.
7774                  */
7775                 dnlc_remove(ndvp, nnm);

7777                 rp = VTOR4(nvp);

7779                 if (nvp->v_type != VREG) {
7780                         /*
```

```
7781                         * Purge the name cache of all references to this vnode
7782                         * so that we can check the reference count to infer
7783                         * whether it is active or not.
7784                         */
7785                        if (nvp->v_count > 1)
7786                                dnlc_purge_vp(nvp);
7787
7788                        isactive = nvp->v_count > 1;
7789                } else {
7790                        mutex_enter(&rp->r_os_lock);
7791                        isactive = list_head(&rp->r_open_streams) != NULL;
7792                        mutex_exit(&rp->r_os_lock);
7793                }
7794
7795                /*
7796                 * If the vnode is active and is not a directory,
7797                 * arrange to rename it to a
7798                 * temporary file so that it will continue to be
7799                 * accessible.  This implements the "unlink-open-file"
7800                 * semantics for the target of a rename operation.
7801                 * Before doing this though, make sure that the
7802                 * source and target files are not already the same.
7803                 */
7804                if (isactive && nvp->v_type != VDIR) {
7805                        /*
7806                         * Lookup the source name.
7807                         */
7808                        error = nfs4lookup(odvp, onm, &ovp, cr, 0);
7809
7810                        /*
7811                         * The source name *should* already exist.
7812                         */
7813                        if (error) {
7814                                VN_RELE(nvp);
7815                                nfs_rw_exit(&odrp->r_rwlock);
7816                                nfs_rw_exit(&ndrp->r_rwlock);
7817                                return (error);
7818                        }
7819
7820                        ASSERT(nfs4_consistent_type(ovp));
7821
7822                        /*
7823                         * Compare the two vnodes.  If they are the same,
7824                         * just release all held vnodes and return success.
7825                         */
7826                        if (VN_CMP(ovp, nvp)) {
7827                                VN_RELE(ovp);
7828                                VN_RELE(nvp);
7829                                nfs_rw_exit(&odrp->r_rwlock);
7830                                nfs_rw_exit(&ndrp->r_rwlock);
7831                                return (0);
7832                        }
7833
7834                        /*
7835                         * Can't mix and match directories and non-
7836                         * directories in rename operations.  We already
7837                         * know that the target is not a directory.  If
7838                         * the source is a directory, return an error.
7839                         */
7840                        if (ovp->v_type == VDIR) {
7841                                VN_RELE(ovp);
7842                                VN_RELE(nvp);
7843                                nfs_rw_exit(&odrp->r_rwlock);
7844                                nfs_rw_exit(&ndrp->r_rwlock);
7845                                return (ENOTDIR);
7846                        }
```



```
7847 link_call:
7848                         /*
7849                          * The target file exists, is not the same as
7850                          * the source file, and is active.  We first
7851                          * try to Link it to a temporary filename to
7852                          * avoid having the server removing the file
7853                          * completely (which could cause data loss to
7854                          * the user's POV in the event the Rename fails
7855                          * -- see bug 1165874).
7856                          */
7857                         /*
7858                          * The do_link and did_link booleans are
7859                          * introduced in the event we get NFS4ERR_FILE_OPEN
7860                          * returned for the Rename.  Some servers can
7861                          * not Rename over an Open file, so they return
7862                          * this error.  The client needs to Remove the
7863                          * newly created Link and do two Renames, just
7864                          * as if the server didn't support LINK.
7865                          */
7866                         tmpname = newname();
7867                         error = 0;
7868
7869                         if (do_link) {
7870                                 error = nfs4_link(ndvp, nvp, tmpname, cr,
7871                                     NULL, 0);
7872                         }
7873                         if (error == EOPNOTSUPP || !do_link) {
7874                                 error = nfs4_rename(ndvp, nnm, ndvp, tmpname,
7875                                     cr, NULL, 0);
7876                                 did_link = 0;
7877                         } else {
7878                                 did_link = 1;
7879                         }
7880                         if (error) {
7881                                 kmem_free(tmpname, MAXNAMELEN);
7882                                 VN_RELE(ovp);
7883                                 VN_RELE(nvp);
7884                                 nfs_rw_exit(&odrp->r_rwlock);
7885                                 nfs_rw_exit(&ndrp->r_rwlock);
7886                                 return (error);
7887                         }
7888
7889                         mutex_enter(&rp->r_statelock);
7890                         if (rp->r_unldvp == NULL) {
7891                                 VN_HOLD(ndvp);
7892                                 rp->r_unldvp = ndvp;
7893                                 if (rp->r_unlcred != NULL)
7894                                         crfree(rp->r_unlcred);
7895                                 crhold(cr);
7896                                 rp->r_unlcred = cr;
7897                                 rp->r_unlname = tmpname;
7898                         } else {
7899                                 if (rp->r_unlname)
7900                                         kmem_free(rp->r_unlname, MAXNAMELEN);
7901                                 rp->r_unlname = tmpname;
7902                         }
7903                         mutex_exit(&rp->r_statelock);
7904                 }
7905
7906                 (void) nfs4delegreturn(VTOR4(nvp), NFS4_DR_PUSH|NFS4_DR_REOPEN);
7907
7908                 ASSERT(nfs4_consistent_type(nvp));
7909         }
7910
7911         if (ovp == NULL) {
7912                 /*
```

```
7913                 * When renaming directories to be a subdirectory of a
7914                 * different parent, the dnlc entry for ".." will no
7915                 * longer be valid, so it must be removed.
7916                 *
7917                 * We do a lookup here to determine whether we are renaming
7918                 * a directory and we need to check if we are renaming
7919                 * an unlinked file.  This might have already been done
7920                 * in previous code, so we check ovp == NULL to avoid
7921                 * doing it twice.
7922                 */
7923                error = nfs4lookup(odvp, onm, &ovp, cr, 0);
7924                /*
7925                 * The source name *should* already exist.
7926                 */
7927                if (error) {
7928                        nfs_rw_exit(&odrp->r_rwlock);
7929                        nfs_rw_exit(&ndrp->r_rwlock);
7930                        if (nvp) {
7931                                VN_RELE(nvp);
7932                        }
7933                        return (error);
7934                }
7935                ASSERT(ovp != NULL);
7936                ASSERT(nfs4_consistent_type(ovp));
7937        }

7939        /*
7940         * Is the object being renamed a dir, and if so, is
7941         * it being renamed to a child of itself?  The underlying
7942         * fs should ultimately return EINVAL for this case;
7943         * however, buggy beta non-Solaris NFSv4 servers at
7944         * interop testing events have allowed this behavior,
7945         * and it caused our client to panic due to a recursive
7946         * mutex_enter in fn_move.
7947         *
7948         * The tedious locking in fn_move could be changed to
7949         * deal with this case, and the client could avoid the
7950         * panic; however, the client would just confuse itself
7951         * later and misbehave.  A better way to handle the broken
7952         * server is to detect this condition and return EINVAL
7953         * without ever sending the the bogus rename to the server.
7954         * We know the rename is invalid -- just fail it now.
7955         */
7956        if (ovp->v_type == VDIR && VN_CMP(ndvp, ovp)) {
7957                VN_RELE(ovp);
7958                nfs_rw_exit(&odrp->r_rwlock);
7959                nfs_rw_exit(&ndrp->r_rwlock);
7960                if (nvp) {
7961                        VN_RELE(nvp);
7962                }
7963                return (EINVAL);
7964        }

7966        (void) nfs4delegreturn(VTOR4(ovp), NFS4_DR_PUSH|NFS4_DR_REOPEN);

7968        /*
7969         * If FH4_VOL_RENAME or FH4_VOLATILE_ANY bits are set, it is
7970         * possible for the filehandle to change due to the rename.
7971         * If neither of these bits is set, but FH4_VOL_MIGRATION is set,
7972         * the fh will not change because of the rename, but we still need
7973         * to update its rnode entry with the new name for
7974         * an eventual fh change due to migration. The FH4_NOEXPIRE_ON_OPEN
7975         * has no effect on these for now, but for future improvements,
7976         * we might want to use it too to simplify handling of files
7977         * that are open with that flag on. (XXX)
7978         */
```



```
7979        mi = VTOMI4(odvp);
7980        if (NFS4_VOLATILE_FH(mi))
7981                error = nfs4rename_volatile_fh(odvp, onm, ovp, ndvp, nnm, cr,
7982                    &stat);
7983        else
7984                error = nfs4rename_persistent_fh(odvp, onm, ovp, ndvp, nnm, cr,
7985                    &stat);

7987        ASSERT(nfs4_consistent_type(odvp));
7988        ASSERT(nfs4_consistent_type(ndvp));
7989        ASSERT(nfs4_consistent_type(ovp));

7991        if (stat == NFS4ERR_FILE_OPEN && did_link) {
7992                do_link = 0;
7993                /*
7994                 * Before the 'link_call' code, we did a nfs4_lookup
7995                 * that puts a VN_HOLD on nvp.  After the nfs4_link
7996                 * call we call VN_RELE to match that hold.  We need
7997                 * to place an additional VN_HOLD here since we will
7998                 * be hitting that VN_RELE again.
7999                 */
8000                VN_HOLD(nvp);

8002                (void) nfs4_remove(ndvp, tmpname, cr, NULL, 0);

8004                /* Undo the unlinked file naming stuff we just did */
8005                mutex_enter(&rp->r_statelock);
8006                if (rp->r_unldvp) {
8007                        VN_RELE(ndvp);
8008                        rp->r_unldvp = NULL;
8009                        if (rp->r_unlcred != NULL)
8010                                crfree(rp->r_unlcred);
8011                        rp->r_unlcred = NULL;
8012                        /* rp->r_unlanme points to tmpname */
8013                        if (rp->r_unlname)
8014                                kmem_free(rp->r_unlname, MAXNAMELEN);
8015                        rp->r_unlname = NULL;
8016                }
8017                mutex_exit(&rp->r_statelock);

8019                if (nvp) {
8020                        VN_RELE(nvp);
8021                }
8022                goto link_call;
8023        }

8025        if (error) {
8026                VN_RELE(ovp);
8027                nfs_rw_exit(&odrp->r_rwlock);
8028                nfs_rw_exit(&ndrp->r_rwlock);
8029                if (nvp) {
8030                        VN_RELE(nvp);
8031                }
8032                return (error);
8033        }

8035        /*
8036         * when renaming directories to be a subdirectory of a
8037         * different parent, the dnlc entry for ".." will no
8038         * longer be valid, so it must be removed
8039         */
8040        rp = VTOR4(ovp);
8041        if (ndvp != odvp) {
8042                if (ovp->v_type == VDIR) {
8043                        dnlc_remove(ovp, "..");
8044                        if (rp->r_dir != NULL)
```

```
8045                                nfs4_purge_rddir_cache(ovp);
8046                }
8047        }

8049        /*
8050         * If we are renaming the unlinked file, update the
8051         * r_unldvp and r_unlname as needed.
8052         */
8053        mutex_enter(&rp->r_statelock);
8054        if (rp->r_unldvp != NULL) {
8055                if (strcmp(rp->r_unlname, onm) == 0) {
8056                        (void) strncpy(rp->r_unlname, nnm, MAXNAMELEN);
8057                        rp->r_unlname[MAXNAMELEN - 1] = '\0';
8058                        if (ndvp != rp->r_unldvp) {
8059                                VN_RELE(rp->r_unldvp);
8060                                rp->r_unldvp = ndvp;
8061                                VN_HOLD(ndvp);
8062                        }
8063                }
8064        }
8065        mutex_exit(&rp->r_statelock);

8067        /*
8068         * Notify the rename vnevents to source vnode, and to the target
8069         * vnode if it already existed.
8070         */
8071        if (error == 0) {
8072                vnode_t *tvp;
8073                rnode4_t *trp;
8074                /*
8075                 * Notify the vnode. Each links is represented by
8076                 * a different vnode, in nfsv4.
8077                 */
8078                if (nvp) {
8079                        trp = VTOR4(nvp);
8080                        tvp = nvp;
8081                        if (IS_SHADOW(nvp, trp))
8082                                tvp = RTOV4(trp);
8083                        vnevent_rename_dest(tvp, ndvp, nnm, ct);
8084                }

8086                /*
8087                 * if the source and destination directory are not the
8088                 * same notify the destination directory.
8089                 */
8090                if (VTOR4(odvp) != VTOR4(ndvp)) {
8091                        trp = VTOR4(ndvp);
8092                        tvp = ndvp;
8093                        if (IS_SHADOW(ndvp, trp))
8094                                tvp = RTOV4(trp);
8095                        vnevent_rename_dest_dir(tvp, ct);
8096                }

8098                trp = VTOR4(ovp);
8099                tvp = ovp;
8100                if (IS_SHADOW(ovp, trp))
8101                        tvp = RTOV4(trp);
8102                vnevent_rename_src(tvp, odvp, onm, ct);
8103        }

8105        if (nvp) {
8106                VN_RELE(nvp);
8107        }
8108        VN_RELE(ovp);

8110        nfs_rw_exit(&odrp->r_rwlock);
```



```
8111        nfs_rw_exit(&ndrp->r_rwlock);

8113        return (error);
8114 }

8116 /*
8117  * When the parent directory has changed, sv_dfh must be updated
8118  */
8119 static void
8120 update_parentdir_sfh(vnode_t *vp, vnode_t *ndvp)
8121 {
8122         svnode_t *sv = VTOSV(vp);
8123         nfs4_sharedfh_t *old_dfh = sv->sv_dfh;
8124         nfs4_sharedfh_t *new_dfh = VTOR4(ndvp)->r_fh;

8126         sfh4_hold(new_dfh);
8127         sv->sv_dfh = new_dfh;
8128         sfh4_rele(&old_dfh);
8129 }

8131 /*
8132  * nfs4rename_persistent does the otw portion of renaming in NFS Version 4,
8133  * when it is known that the filehandle is persistent through rename.
8134  *
8135  * Rename requires that the current fh be the target directory and the
8136  * saved fh be the source directory. After the operation, the current fh
8137  * is unchanged.
8138  * The compound op structure for persistent fh rename is:
8139  *      PUTFH(sourcdir), SAVEFH, PUTFH(targetdir), RENAME
8140  * Rather than bother with the directory postop args, we'll simply
8141  * update that a change occurred in the cache, so no post-op getattrs.
8142  */
8143 static int
8144 nfs4rename_persistent_fh(vnode_t *odvp, char *onm, vnode_t *renvp,
8145     vnode_t *ndvp, char *nnm, cred_t *cr, nfsstat4 *statp)
8146 {
8147         COMPOUND4args_clnt args;
8148         COMPOUND4res_clnt res, *resp = NULL;
8149         nfs_argop4 *argop;
8150         nfs_resop4 *resop;
8151         int doqueue, argoplist_size;
8152         mntinfo4_t *mi;
8153         rnode4_t *odrp = VTOR4(odvp);
8154         rnode4_t *ndrp = VTOR4(ndvp);
8155         RENAME4res *rn_res;
8156         bool_t needrecov;
8157         nfs4_recov_state_t recov_state;
8158         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
8159         dirattr_info_t dinfo, *dinfop;

8161         ASSERT(nfs_zone() == VTOMI4(odvp)->mi_zone);

8163         recov_state.rs_flags = 0;
8164         recov_state.rs_num_retry_despite_err = 0;

8166         /*
8167          * Rename ops: putfh sdir; savefh; putfh tdir; rename; getattr tdir
8168          *
8169          * If source/target are different dirs, then append putfh(src); getattr
8170          */
8171         args.array_len = (odvp == ndvp) ? 5 : 7;
8172         argoplist_size = args.array_len * sizeof (nfs_argop4);
8173         args.array = argop = kmem_alloc(argoplist_size, KM_SLEEP);

8175 recov_retry:
8176         *statp = NFS4_OK;
```

```
8178         /* No need to Lookup the file, persistent fh */
8179         args.ctag = TAG_RENAME;

8181         mi = VTOMI4(odvp);
8182         e.error = nfs4_start_op(mi, odvp, ndvp, &recov_state);
8183         if (e.error) {
8184                 kmem_free(argop, argoplist_size);
8185                 return (e.error);
8186         }

8188         /* 0: putfh source directory */
8189         argop[0].argop = OP_CPUTFH;
8190         argop[0].nfs_argop4_u.opcputfh.sfh = odrp->r_fh;

8192         /* 1: Save source fh to free up current for target */
8193         argop[1].argop = OP_SAVEFH;

8195         /* 2: putfh targetdir */
8196         argop[2].argop = OP_CPUTFH;
8197         argop[2].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;

8199         /* 3: current_fh is targetdir, saved_fh is sourcedir */
8200         argop[3].argop = OP_CRENAME;
8201         argop[3].nfs_argop4_u.opcrename.coldname = onm;
8202         argop[3].nfs_argop4_u.opcrename.cnewname = nnm;

8204         /* 4: getattr (targetdir) */
8205         argop[4].argop = OP_GETATTR;
8206         argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8207         argop[4].nfs_argop4_u.opgetattr.mi = mi;

8209         if (ndvp != odvp) {

8211                 /* 5: putfh (sourcedir) */
8212                 argop[5].argop = OP_CPUTFH;
8213                 argop[5].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;

8215                 /* 6: getattr (sourcedir) */
8216                 argop[6].argop = OP_GETATTR;
8217                 argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8218                 argop[6].nfs_argop4_u.opgetattr.mi = mi;
8219         }

8221         dnlc_remove(odvp, onm);
8222         dnlc_remove(ndvp, nnm);

8224         doqueue = 1;
8225         dinfo.di_time_call = gethrtime();
8226         rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

8228         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
8229         if (e.error) {
8230                 PURGE_ATTRCACHE4(odvp);
8231                 PURGE_ATTRCACHE4(ndvp);
8232         } else {
8233                 *statp = res.status;
8234         }

8236         if (needrecov) {
8237                 if (nfs4_start_recovery(&e, mi, odvp, ndvp, NULL, NULL,
8238                     OP_RENAME, NULL, NULL, NULL) == FALSE) {
8239                         nfs4_end_op(mi, odvp, ndvp, &recov_state, needrecov);
8240                         if (!e.error)
8241                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
8242                                     (caddr_t)&res);
```



```
8243                         goto recov_retry;
8244                 }
8245         }

8247         if (!e.error) {
8248                 resp = &res;
8249                 /*
8250                  * as long as OP_RENAME
8251                  */
8252                 if (res.status != NFS4_OK && res.array_len <= 4) {
8253                         e.error = geterrno4(res.status);
8254                         PURGE_ATTRCACHE4(odvp);
8255                         PURGE_ATTRCACHE4(ndvp);
8256                         /*
8257                          * System V defines rename to return EEXIST, not
8258                          * ENOTEMPTY if the target directory is not empty.
8259                          * Over the wire, the error is NFSERR_ENOTEMPTY
8260                          * which geterrno4 maps to ENOTEMPTY.
8261                          */
8262                         if (e.error == ENOTEMPTY)
8263                                 e.error = EEXIST;
8264                 } else {

8266                         resop = &res.array[3];  /* rename res */
8267                         rn_res = &resop->nfs_resop4_u.oprename;

8269                         if (res.status == NFS4_OK) {
8270                                 /*
8271                                  * Update target attribute, readdir and dnlc
8272                                  * caches.
8273                                  */
8274                                 dinfo.di_garp =
8275                                     &res.array[4].nfs_resop4_u.opgetattr.ga_res;
8276                                 dinfo.di_cred = cr;
8277                                 dinfop = &dinfo;
8278                         } else
8279                                 dinfop = NULL;

8281                         nfs4_update_dircaches(&rn_res->target_cinfo,
8282                             ndvp, NULL, NULL, dinfop);

8284                         /*
8285                          * Update source attribute, readdir and dnlc caches
8286                          *
8287                          */
8288                         if (ndvp != odvp) {
8289                                 update_parentdir_sfh(renvp, ndvp);

8291                                 if (dinfop)
8292                                         dinfo.di_garp =
8293                                             &(res.array[6].nfs_resop4_u.
8294                                             opgetattr.ga_res);

8296                                 nfs4_update_dircaches(&rn_res->source_cinfo,
8297                                     odvp, NULL, NULL, dinfop);
8298                         }

8300                         fn_move(VTOSV(renvp)->sv_name, VTOSV(ndvp)->sv_name,
8301                             nnm);
8302                 }
8303         }

8305         if (resp)
8306                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
8307         nfs4_end_op(mi, odvp, ndvp, &recov_state, needrecov);
8308         kmem_free(argop, argoplist_size);
```

```
8310         return (e.error);
8311 }

8313 /*
8314  * nfs4rename_volatile_fh does the otw part of renaming in NFS Version 4, when
8315  * it is possible for the filehandle to change due to the rename.
8316  *
8317  * The compound req in this case includes a post-rename lookup and getattr
8318  * to ensure that we have the correct fh and attributes for the object.
8319  *
8320  * Rename requires that the current fh be the target directory and the
8321  * saved fh be the source directory. After the operation, the current fh
8322  * is unchanged.
8323  *
8324  * We need the new filehandle (hence a LOOKUP and GETFH) so that we can
8325  * update the filehandle for the renamed object.  We also get the old
8326  * filehandle for historical reasons; this should be taken out sometime.
8327  * This results in a rather cumbersome compound...
8328  *
8329  *    PUTFH(sourcdir), SAVEFH, LOOKUP(src), GETFH(old),
8330  *    PUTFH(targetdir), RENAME, LOOKUP(trgt), GETFH(new), GETATTR
8331  *
8332  */
8333 static int
8334 nfs4rename_volatile_fh(vnode_t *odvp, char *onm, vnode_t *ovp,
8335     vnode_t *ndvp, char *nnm, cred_t *cr, nfsstat4 *statp)
8336 {
8337         COMPOUND4args_clnt args;
8338         COMPOUND4res_clnt res, *resp = NULL;
8339         int argoplist_size;
8340         nfs_argop4 *argop;
8341         nfs_resop4 *resop;
8342         int doqueue;
8343         mntinfo4_t *mi;
8344         rnode4_t *odrp = VTOR4(odvp);   /* old directory */
8345         rnode4_t *ndrp = VTOR4(ndvp);   /* new directory */
8346         rnode4_t *orp = VTOR4(ovp);     /* object being renamed */
8347         RENAME4res *rn_res;
8348         GETFH4res *ngf_res;
8349         bool_t needrecov;
8350         nfs4_recov_state_t recov_state;
8351         hrtime_t t;
8352         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
8353         dirattr_info_t dinfo, *dinfop = &dinfo;

8355         ASSERT(nfs_zone() == VTOMI4(odvp)->mi_zone);

8357         recov_state.rs_flags = 0;
8358         recov_state.rs_num_retry_despite_err = 0;

8360 recov_retry:
8361         *statp = NFS4_OK;

8363         /*
8364          * There is a window between the RPC and updating the path and
8365          * filehandle stored in the rnode.  Lock out the FHEXPIRED recovery
8366          * code, so that it doesn't try to use the old path during that
8367          * window.
8368          */
8369         mutex_enter(&orp->r_statelock);
8370         while (orp->r_flags & R4RECEXPFH) {
8371                 klwp_t *lwp = ttolwp(curthread);

8373                 if (lwp != NULL)
8374                         lwp->lwp_nostop++;
```



```
8375                 if (cv_wait_sig(&orp->r_cv, &orp->r_statelock) == 0) {
8376                         mutex_exit(&orp->r_statelock);
8377                         if (lwp != NULL)
8378                                 lwp->lwp_nostop--;
8379                         return (EINTR);
8380                 }
8381                 if (lwp != NULL)
8382                         lwp->lwp_nostop--;
8383         }
8384         orp->r_flags |= R4RECEXPFH;
8385         mutex_exit(&orp->r_statelock);

8387         mi = VTOMI4(odvp);

8389         args.ctag = TAG_RENAME_VFH;
8390         args.array_len = (odvp == ndvp) ? 10 : 12;
8391         argoplist_size  = args.array_len * sizeof (nfs_argop4);
8392         argop = kmem_alloc(argoplist_size, KM_SLEEP);

8394         /*
8395          * Rename ops:
8396          *    PUTFH(sourcdir), SAVEFH, LOOKUP(src), GETFH(old),
8397          *    PUTFH(targetdir), RENAME, GETATTR(targetdir)
8398          *    LOOKUP(trgt), GETFH(new), GETATTR,
8399          *
8400          *    if (odvp != ndvp)
8401          *      add putfh(sourcedir), getattr(sourcedir) }
8402          */
8403         args.array = argop;

8405         e.error = nfs4_start_fop(mi, odvp, ndvp, OH_VFH_RENAME,
8406             &recov_state, NULL);
8407         if (e.error) {
8408                 kmem_free(argop, argoplist_size);
8409                 mutex_enter(&orp->r_statelock);
8410                 orp->r_flags &= ~R4RECEXPFH;
8411                 cv_broadcast(&orp->r_cv);
8412                 mutex_exit(&orp->r_statelock);
8413                 return (e.error);
8414         }

8416         /* 0: putfh source directory */
8417         argop[0].argop = OP_CPUTFH;
8418         argop[0].nfs_argop4_u.opcputfh.sfh = odrp->r_fh;

8420         /* 1: Save source fh to free up current for target */
8421         argop[1].argop = OP_SAVEFH;

8423         /* 2: Lookup pre-rename fh of renamed object */
8424         argop[2].argop = OP_CLOOKUP;
8425         argop[2].nfs_argop4_u.opclookup.cname = onm;

8427         /* 3: getfh fh of renamed object (before rename) */
8428         argop[3].argop = OP_GETFH;

8430         /* 4: putfh targetdir */
8431         argop[4].argop = OP_CPUTFH;
8432         argop[4].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;

8434         /* 5: current_fh is targetdir, saved_fh is sourcedir */
8435         argop[5].argop = OP_CRENAME;
8436         argop[5].nfs_argop4_u.opcrename.coldname = onm;
8437         argop[5].nfs_argop4_u.opcrename.cnewname = nnm;

8439         /* 6: getattr of target dir (post op attrs) */
8440         argop[6].argop = OP_GETATTR;
```

```
8441         argop[6].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8442         argop[6].nfs_argop4_u.opgetattr.mi = mi;
8443
8444         /* 7: Lookup post-rename fh of renamed object */
8445         argop[7].argop = OP_CLOOKUP;
8446         argop[7].nfs_argop4_u.opclookup.cname = nnm;
8447
8448         /* 8: getfh fh of renamed object (after rename) */
8449         argop[8].argop = OP_GETFH;
8450
8451         /* 9: getattr of renamed object */
8452         argop[9].argop = OP_GETATTR;
8453         argop[9].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8454         argop[9].nfs_argop4_u.opgetattr.mi = mi;
8455
8456         /*
8457          * If source/target dirs are different, then get new post-op
8458          * attrs for source dir also.
8459          */
8460         if (ndvp != odvp) {
8461                 /* 10: putfh (sourcedir) */
8462                 argop[10].argop = OP_CPUTFH;
8463                 argop[10].nfs_argop4_u.opcputfh.sfh = ndrp->r_fh;
8464
8465                 /* 11: getattr (sourcedir) */
8466                 argop[11].argop = OP_GETATTR;
8467                 argop[11].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8468                 argop[11].nfs_argop4_u.opgetattr.mi = mi;
8469         }
8470
8471         dnlc_remove(odvp, onm);
8472         dnlc_remove(ndvp, nnm);
8473
8474         doqueue = 1;
8475         t = gethrtime();
8476         rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);
8477
8478         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
8479         if (e.error) {
8480                 PURGE_ATTRCACHE4(odvp);
8481                 PURGE_ATTRCACHE4(ndvp);
8482                 if (!needrecov) {
8483                         nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME,
8484                             &recov_state, needrecov);
8485                         goto out;
8486                 }
8487         } else {
8488                 *statp = res.status;
8489         }
8490
8491         if (needrecov) {
8492                 bool_t abort;
8493
8494                 abort = nfs4_start_recovery(&e, mi, odvp, ndvp, NULL, NULL,
8495                     OP_RENAME, NULL, NULL, NULL);
8496                 if (abort == FALSE) {
8497                         nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME,
8498                             &recov_state, needrecov);
8499                         kmem_free(argop, argoplist_size);
8500                         if (!e.error)
8501                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
8502                                     (caddr_t)&res);
8503                         mutex_enter(&orp->r_statelock);
8504                         orp->r_flags &= ~R4RECEXPFH;
8505                         cv_broadcast(&orp->r_cv);
8506                         mutex_exit(&orp->r_statelock);
```



```
8507                         goto recov_retry;
8508                 } else {
8509                         if (e.error != 0) {
8510                                 nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME,
8511                                     &recov_state, needrecov);
8512                                 goto out;
8513                         }
8514                         /* fall through for res.status case */
8515                 }
8516         }
8517
8518         resp = &res;
8519         /*
8520          * If OP_RENAME (or any prev op) failed, then return an error.
8521          * OP_RENAME is index 5, so if array len <= 6 we return an error.
8522          */
8523         if ((res.status != NFS4_OK) && (res.array_len <= 6)) {
8524                 /*
8525                  * Error in an op other than last Getattr
8526                  */
8527                 e.error = geterrno4(res.status);
8528                 PURGE_ATTRCACHE4(odvp);
8529                 PURGE_ATTRCACHE4(ndvp);
8530                 /*
8531                  * System V defines rename to return EEXIST, not
8532                  * ENOTEMPTY if the target directory is not empty.
8533                  * Over the wire, the error is NFSERR_ENOTEMPTY
8534                  * which geterrno4 maps to ENOTEMPTY.
8535                  */
8536                 if (e.error == ENOTEMPTY)
8537                         e.error = EEXIST;
8538                 nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME, &recov_state,
8539                     needrecov);
8540                 goto out;
8541         }
8542
8543         /* rename results */
8544         rn_res = &res.array[5].nfs_resop4_u.oprename;
8545
8546         if (res.status == NFS4_OK) {
8547                 /* Update target attribute, readdir and dnlc caches */
8548                 dinfo.di_garp =
8549                     &res.array[6].nfs_resop4_u.opgetattr.ga_res;
8550                 dinfo.di_cred = cr;
8551                 dinfo.di_time_call = t;
8552         } else
8553                 dinfop = NULL;
8554
8555         /* Update source cache attribute, readdir and dnlc caches */
8556         nfs4_update_dircaches(&rn_res->target_cinfo, ndvp, NULL, NULL, dinfop);
8557
8558         /* Update source cache attribute, readdir and dnlc caches */
8559         if (ndvp != odvp) {
8560                 update_parentdir_sfh(ovp, ndvp);
8561
8562                 /*
8563                  * If dinfop is non-NULL, then compound succeded, so
8564                  * set di_garp to attrs for source dir.  dinfop is only
8565                  * set to NULL when compound fails.
8566                  */
8567                 if (dinfop)
8568                         dinfo.di_garp =
8569                             &res.array[11].nfs_resop4_u.opgetattr.ga_res;
8570                 nfs4_update_dircaches(&rn_res->source_cinfo, odvp, NULL, NULL,
8571                     dinfop);
8572         }
```

```
8574         /*
8575          * Update the rnode with the new component name and args,
8576          * and if the file handle changed, also update it with the new fh.
8577          * This is only necessary if the target object has an rnode
8578          * entry and there is no need to create one for it.
8579          */
8580         resop = &res.array[8];  /* getfh new res */
8581         ngf_res = &resop->nfs_resop4_u.opgetfh;

8583         /*
8584          * Update the path and filehandle for the renamed object.
8585          */
8586         nfs4rename_update(ovp, ndvp, &ngf_res->object, nnm);

8588         nfs4_end_fop(mi, odvp, ndvp, OH_VFH_RENAME, &recov_state, needrecov);

8590         if (res.status == NFS4_OK) {
8591                 resop++;        /* getattr res */
8592                 e.error = nfs4_update_attrcache(res.status,
8593                     &resop->nfs_resop4_u.opgetattr.ga_res,
8594                     t, ovp, cr);
8595         }

8597 out:
8598         kmem_free(argop, argoplist_size);
8599         if (resp)
8600                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
8601         mutex_enter(&orp->r_statelock);
8602         orp->r_flags &= ~R4RECEXPFH;
8603         cv_broadcast(&orp->r_cv);
8604         mutex_exit(&orp->r_statelock);

8606         return (e.error);
8607 }

8609 /* ARGSUSED */
8610 static int
8611 nfs4_mkdir(vnode_t *dvp, char *nm, struct vattr *va, vnode_t **vpp, cred_t *cr,
8612     caller_context_t *ct, int flags, vsecattr_t *vsecp)
8613 {
8614         int error;
8615         vnode_t *vp;

8617         if (nfs_zone() != VTOMI4(dvp)->mi_zone)
8618                 return (EPERM);
8619         /*
8620          * As ".." has special meaning and rather than send a mkdir
8621          * over the wire to just let the server freak out, we just
8622          * short circuit it here and return EEXIST
8623          */
8624         if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0')
8625                 return (EEXIST);

8627         /*
8628          * Decision to get the right gid and setgid bit of the
8629          * new directory is now made in call_nfs4_create_req.
8630          */
8631         va->va_mask |= AT_MODE;
8632         error = call_nfs4_create_req(dvp, nm, NULL, va, &vp, cr, NF4DIR);
8633         if (error)
8634                 return (error);

8636         *vpp = vp;
8637         return (0);
8638 }
```



```
8641 /*
8642  * rmdir is using the same remove v4 op as does remove.
8643  * Remove requires that the current fh be the target directory.
8644  * After the operation, the current fh is unchanged.
8645  * The compound op structure is:
8646  *      PUTFH(targetdir), REMOVE
8647  */
8648 /*ARGSUSED4*/
8649 static int
8650 nfs4_rmdir(vnode_t *dvp, char *nm, vnode_t *cdir, cred_t *cr,
8651     caller_context_t *ct, int flags)
8652 {
8653         int need_end_op = FALSE;
8654         COMPOUND4args_clnt args;
8655         COMPOUND4res_clnt res, *resp = NULL;
8656         REMOVE4res *rm_res;
8657         nfs_argop4 argop[3];
8658         nfs_resop4 *resop;
8659         vnode_t *vp;
8660         int doqueue;
8661         mntinfo4_t *mi;
8662         rnode4_t *drp;
8663         bool_t needrecov = FALSE;
8664         nfs4_recov_state_t recov_state;
8665         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };
8666         dirattr_info_t dinfo, *dinfop;

8668         if (nfs_zone() != VTOMI4(dvp)->mi_zone)
8669                 return (EPERM);
8670         /*
8671          * As ".." has special meaning and rather than send a rmdir
8672          * over the wire to just let the server freak out, we just
8673          * short circuit it here and return EEXIST
8674          */
8675         if (nm[0] == '.' && nm[1] == '.' && nm[2] == '\0')
8676                 return (EEXIST);

8678         drp = VTOR4(dvp);
8679         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_WRITER, INTR4(dvp)))
8680                 return (EINTR);

8682         /*
8683          * Attempt to prevent a rmdir(".") from succeeding.
8684          */
8685         e.error = nfs4lookup(dvp, nm, &vp, cr, 0);
8686         if (e.error) {
8687                 nfs_rw_exit(&drp->r_rwlock);
8688                 return (e.error);
8689         }
8690         if (vp == cdir) {
8691                 VN_RELE(vp);
8692                 nfs_rw_exit(&drp->r_rwlock);
8693                 return (EINVAL);
8694         }

8696         /*
8697          * Since nfsv4 remove op works on both files and directories,
8698          * check that the removed object is indeed a directory.
8699          */
8700         if (vp->v_type != VDIR) {
8701                 VN_RELE(vp);
8702                 nfs_rw_exit(&drp->r_rwlock);
8703                 return (ENOTDIR);
8704         }
```

```
8706         /*
8707          * First just remove the entry from the name cache, as it
8708          * is most likely an entry for this vp.
8709          */
8710         dnlc_remove(dvp, nm);

8712         /*
8713          * If there vnode reference count is greater than one, then
8714          * there may be additional references in the DNLC which will
8715          * need to be purged.  First, trying removing the entry for
8716          * the parent directory and see if that removes the additional
8717          * reference(s).  If that doesn't do it, then use dnlc_purge_vp
8718          * to completely remove any references to the directory which
8719          * might still exist in the DNLC.
8720          */
8721         if (vp->v_count > 1) {
8722                 dnlc_remove(vp, "..");
8723                 if (vp->v_count > 1)
8724                         dnlc_purge_vp(vp);
8725         }

8727         mi = VTOMI4(dvp);
8728         recov_state.rs_flags = 0;
8729         recov_state.rs_num_retry_despite_err = 0;

8731 recov_retry:
8732         args.ctag = TAG_RMDIR;

8734         /*
8735          * Rmdir ops: putfh dir; remove
8736          */
8737         args.array_len = 3;
8738         args.array = argop;

8740         e.error = nfs4_start_op(VTOMI4(dvp), dvp, NULL, &recov_state);
8741         if (e.error) {
8742                 nfs_rw_exit(&drp->r_rwlock);
8743                 return (e.error);
8744         }
8745         need_end_op = TRUE;

8747         /* putfh directory */
8748         argop[0].argop = OP_CPUTFH;
8749         argop[0].nfs_argop4_u.opcputfh.sfh = drp->r_fh;

8751         /* remove */
8752         argop[1].argop = OP_CREMOVE;
8753         argop[1].nfs_argop4_u.opcremove.ctarget = nm;

8755         /* getattr (postop attrs for dir that contained removed dir) */
8756         argop[2].argop = OP_GETATTR;
8757         argop[2].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
8758         argop[2].nfs_argop4_u.opgetattr.mi = mi;

8760         dinfo.di_time_call = gethrtime();
8761         doqueue = 1;
8762         rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

8764         PURGE_ATTRCACHE4(vp);

8766         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
8767         if (e.error) {
8768                 PURGE_ATTRCACHE4(dvp);
8769         }
```



```
8771         if (needrecov) {
8772                 if (nfs4_start_recovery(&e, VTOMI4(dvp), dvp, NULL, NULL,
8773                     NULL, OP_REMOVE, NULL, NULL, NULL) == FALSE) {
8774                         if (!e.error)
8775                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
8776                                     (caddr_t)&res);

8778                         nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state,
8779                             needrecov);
8780                         need_end_op = FALSE;
8781                         goto recov_retry;
8782                 }
8783         }

8785         if (!e.error) {
8786                 resp = &res;

8788                 /*
8789                  * Only return error if first 2 ops (OP_REMOVE or earlier)
8790                  * failed.
8791                  */
8792                 if (res.status != NFS4_OK && res.array_len <= 2) {
8793                         e.error = geterrno4(res.status);
8794                         PURGE_ATTRCACHE4(dvp);
8795                         nfs4_end_op(VTOMI4(dvp), dvp, NULL,
8796                             &recov_state, needrecov);
8797                         need_end_op = FALSE;
8798                         nfs4_purge_stale_fh(e.error, dvp, cr);
8799                         /*
8800                          * System V defines rmdir to return EEXIST, not
8801                          * ENOTEMPTY if the directory is not empty.  Over
8802                          * the wire, the error is NFSERR_ENOTEMPTY which
8803                          * geterrno4 maps to ENOTEMPTY.
8804                          */
8805                         if (e.error == ENOTEMPTY)
8806                                 e.error = EEXIST;
8807                 } else {
8808                         resop = &res.array[1];  /* remove res */
8809                         rm_res = &resop->nfs_resop4_u.opremove;

8811                         if (res.status == NFS4_OK) {
8812                                 resop = &res.array[2];  /* dir attrs */
8813                                 dinfo.di_garp =
8814                                     &resop->nfs_resop4_u.opgetattr.ga_res;
8815                                 dinfo.di_cred = cr;
8816                                 dinfop = &dinfo;
8817                         } else
8818                                 dinfop = NULL;

8820                         /* Update dir attribute, readdir and dnlc caches */
8821                         nfs4_update_dircaches(&rm_res->cinfo, dvp, NULL, NULL,
8822                             dinfop);

8824                         /* destroy rddir cache for dir that was removed */
8825                         if (VTOR4(vp)->r_dir != NULL)
8826                                 nfs4_purge_rddir_cache(vp);
8827                 }
8828         }

8830         if (need_end_op)
8831                 nfs4_end_op(VTOMI4(dvp), dvp, NULL, &recov_state, needrecov);

8833         nfs_rw_exit(&drp->r_rwlock);

8835         if (resp)
8836                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)resp);
```

```
8838         if (e.error == 0) {
8839                 vnode_t *tvp;
8840                 rnode4_t *trp;
8841                 trp = VTOR4(vp);
8842                 tvp = vp;
8843                 if (IS_SHADOW(vp, trp))
8844                         tvp = RTOV4(trp);
8845                 vnevent_rmdir(tvp, dvp, nm, ct);
8846         }

8848         VN_RELE(vp);

8850         return (e.error);
8851 }

8853 /* ARGSUSED */
8854 static int
8855 nfs4_symlink(vnode_t *dvp, char *lnm, struct vattr *tva, char *tnm, cred_t *cr,
8856     caller_context_t *ct, int flags)
8857 {
8858         int error;
8859         vnode_t *vp;
8860         rnode4_t *rp;
8861         char *contents;
8862         mntinfo4_t *mi = VTOMI4(dvp);

8864         if (nfs_zone() != mi->mi_zone)
8865                 return (EPERM);
8866         if (!(mi->mi_flags & MI4_SYMLINK))
8867                 return (EOPNOTSUPP);

8869         error = call_nfs4_create_req(dvp, lnm, tnm, tva, &vp, cr, NF4LNK);
8870         if (error)
8871                 return (error);

8873         ASSERT(nfs4_consistent_type(vp));
8874         rp = VTOR4(vp);
8875         if (nfs4_do_symlink_cache && rp->r_symlink.contents == NULL) {

8877                 contents = kmem_alloc(MAXPATHLEN, KM_SLEEP);

8879                 if (contents != NULL) {
8880                         mutex_enter(&rp->r_statelock);
8881                         if (rp->r_symlink.contents == NULL) {
8882                                 rp->r_symlink.len = strlen(tnm);
8883                                 bcopy(tnm, contents, rp->r_symlink.len);
8884                                 rp->r_symlink.contents = contents;
8885                                 rp->r_symlink.size = MAXPATHLEN;
8886                                 mutex_exit(&rp->r_statelock);
8887                         } else {
8888                                 mutex_exit(&rp->r_statelock);
8889                                 kmem_free((void *)contents, MAXPATHLEN);
8890                         }
8891                 }
8892         }
8893         VN_RELE(vp);

8895         return (error);
8896 }


8899 /*
8900  * Read directory entries.
8901  * There are some weird things to look out for here.  The uio_loffset
8902  * field is either 0 or it is the offset returned from a previous
```



```
8903  * readdir.  It is an opaque value used by the server to find the
8904  * correct directory block to read. The count field is the number
8905  * of blocks to read on the server.  This is advisory only, the server
8906  * may return only one block's worth of entries.  Entries may be compressed
8907  * on the server.
8908  */
8909 /* ARGSUSED */
8910 static int
8911 nfs4_readdir(vnode_t *vp, struct uio *uiop, cred_t *cr, int *eofp,
8912         caller_context_t *ct, int flags)
8913 {
8914         int error;
8915         uint_t count;
8916         rnode4_t *rp;
8917         rddir4_cache *rdc;
8918         rddir4_cache *rrdc;

8920         if (nfs_zone() != VTOMI4(vp)->mi_zone)
8921                 return (EIO);
8922         rp = VTOR4(vp);

8924         ASSERT(nfs_rw_lock_held(&rp->r_rwlock, RW_READER));

8926         /*
8927          * Make sure that the directory cache is valid.
8928          */
8929         if (rp->r_dir != NULL) {
8930                 if (nfs_disable_rddir_cache != 0) {
8931                         /*
8932                          * Setting nfs_disable_rddir_cache in /etc/system
8933                          * allows interoperability with servers that do not
8934                          * properly update the attributes of directories.
8935                          * Any cached information gets purged before an
8936                          * access is made to it.
8937                          */
8938                         nfs4_purge_rddir_cache(vp);
8939                 }

8941                 error = nfs4_validate_caches(vp, cr);
8942                 if (error)
8943                         return (error);
8944         }

8946         count = MIN(uiop->uio_iov->iov_len, MAXBSIZE);

8948         /*
8949          * Short circuit last readdir which always returns 0 bytes.
8950          * This can be done after the directory has been read through
8951          * completely at least once.  This will set r_direof which
8952          * can be used to find the value of the last cookie.
8953          */
8954         mutex_enter(&rp->r_statelock);
8955         if (rp->r_direof != NULL &&
8956             uiop->uio_loffset == rp->r_direof->nfs4_ncookie) {
8957                 mutex_exit(&rp->r_statelock);
8958 #ifdef DEBUG
8959                 nfs4_readdir_cache_shorts++;
8960 #endif
8961                 if (eofp)
8962                         *eofp = 1;
8963                 return (0);
8964         }

8966         /*
8967          * Look for a cache entry.  Cache entries are identified
8968          * by the NFS cookie value and the byte count requested.
```

```
8969          */
8970         rdc = rddir4_cache_lookup(rp, uiop->uio_loffset, count);

8972         /*
8973          * If rdc is NULL then the lookup resulted in an unrecoverable error.
8974          */
8975         if (rdc == NULL) {
8976                 mutex_exit(&rp->r_statelock);
8977                 return (EINTR);
8978         }

8980         /*
8981          * Check to see if we need to fill this entry in.
8982          */
8983         if (rdc->flags & RDDIRREQ) {
8984                 rdc->flags &= ~RDDIRREQ;
8985                 rdc->flags |= RDDIR;
8986                 mutex_exit(&rp->r_statelock);

8988                 /*
8989                  * Do the readdir.
8990                  */
8991                 nfs4readdir(vp, rdc, cr);

8993                 /*
8994                  * Reacquire the lock, so that we can continue
8995                  */
8996                 mutex_enter(&rp->r_statelock);
8997                 /*
8998                  * The entry is now complete
8999                  */
9000                 rdc->flags &= ~RDDIR;
9001         }

9003         ASSERT(!(rdc->flags & RDDIR));

9005         /*
9006          * If an error occurred while attempting
9007          * to fill the cache entry, mark the entry invalid and
9008          * just return the error.
9009          */
9010         if (rdc->error) {
9011                 error = rdc->error;
9012                 rdc->flags |= RDDIRREQ;
9013                 rddir4_cache_rele(rp, rdc);
9014                 mutex_exit(&rp->r_statelock);
9015                 return (error);
9016         }

9018         /*
9019          * The cache entry is complete and good,
9020          * copyout the dirent structs to the calling
9021          * thread.
9022          */
9023         error = uiomove(rdc->entries, rdc->actlen, UIO_READ, uiop);

9025         /*
9026          * If no error occurred during the copyout,
9027          * update the offset in the uio struct to
9028          * contain the value of the next NFS 4 cookie
9029          * and set the eof value appropriately.
9030          */
9031         if (!error) {
9032                 uiop->uio_loffset = rdc->nfs4_ncookie;
9033                 if (eofp)
9034                         *eofp = rdc->eof;
```



```
9035         }

9037         /*
9038          * Decide whether to do readahead.  Don't if we
9039          * have already read to the end of directory.
9040          */
9041         if (rdc->eof) {
9042                 /*
9043                  * Make the entry the direof only if it is cached
9044                  */
9045                 if (rdc->flags & RDDIRCACHED)
9046                         rp->r_direof = rdc;
9047                 rddir4_cache_rele(rp, rdc);
9048                 mutex_exit(&rp->r_statelock);
9049                 return (error);
9050         }

9052         /* Determine if a readdir readahead should be done */
9053         if (!(rp->r_flags & R4LOOKUP)) {
9054                 rddir4_cache_rele(rp, rdc);
9055                 mutex_exit(&rp->r_statelock);
9056                 return (error);
9057         }

9059         /*
9060          * Now look for a readahead entry.
9061          *
9062          * Check to see whether we found an entry for the readahead.
9063          * If so, we don't need to do anything further, so free the new
9064          * entry if one was allocated.  Otherwise, allocate a new entry, add
9065          * it to the cache, and then initiate an asynchronous readdir
9066          * operation to fill it.
9067          */
9068         rrdc = rddir4_cache_lookup(rp, rdc->nfs4_ncookie, count);

9070         /*
9071          * A readdir cache entry could not be obtained for the readahead.  In
9072          * this case we skip the readahead and return.
9073          */
9074         if (rrdc == NULL) {
9075                 rddir4_cache_rele(rp, rdc);
9076                 mutex_exit(&rp->r_statelock);
9077                 return (error);
9078         }

9080         /*
9081          * Check to see if we need to fill this entry in.
9082          */
9083         if (rrdc->flags & RDDIRREQ) {
9084                 rrdc->flags &= ~RDDIRREQ;
9085                 rrdc->flags |= RDDIR;
9086                 rddir4_cache_rele(rp, rdc);
9087                 mutex_exit(&rp->r_statelock);
9088 #ifdef DEBUG
9089                 nfs4_readdir_readahead++;
9090 #endif
9091                 /*
9092                  * Do the readdir.
9093                  */
9094                 nfs4_async_readdir(vp, rrdc, cr, do_nfs4readdir);
9095                 return (error);
9096         }

9098         rddir4_cache_rele(rp, rrdc);
9099         rddir4_cache_rele(rp, rdc);
9100         mutex_exit(&rp->r_statelock);
```

```
9101         return (error);
9102 }

9104 static int
9105 do_nfs4readdir(vnode_t *vp, rddir4_cache *rdc, cred_t *cr)
9106 {
9107         int error;
9108         rnode4_t *rp;

9110         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

9112         rp = VTOR4(vp);

9114         /*
9115          * Obtain the readdir results for the caller.
9116          */
9117         nfs4readdir(vp, rdc, cr);

9119         mutex_enter(&rp->r_statelock);
9120         /*
9121          * The entry is now complete
9122          */
9123         rdc->flags &= ~RDDIR;

9125         error = rdc->error;
9126         if (error)
9127                 rdc->flags |= RDDIRREQ;
9128         rddir4_cache_rele(rp, rdc);
9129         mutex_exit(&rp->r_statelock);

9131         return (error);
9132 }

9134 /*
9135  * Read directory entries.
9136  * There are some weird things to look out for here.  The uio_loffset
9137  * field is either 0 or it is the offset returned from a previous
9138  * readdir.  It is an opaque value used by the server to find the
9139  * correct directory block to read. The count field is the number
9140  * of blocks to read on the server.  This is advisory only, the server
9141  * may return only one block's worth of entries.  Entries may be compressed
9142  * on the server.
9143  *
9144  * Generates the following compound request:
9145  * 1. If readdir offset is zero and no dnlc entry for parent exists,
9146  *    must include a Lookupp as well. In this case, send:
9147  *    { Putfh <fh>; Readdir; Lookupp; Getfh; Getattr }
9148  * 2. Otherwise just do: { Putfh <fh>; Readdir }
9149  *
9150  * Get complete attributes and filehandles for entries if this is the
9151  * first read of the directory. Otherwise, just get fileid's.
9152  */
9153 static void
9154 nfs4readdir(vnode_t *vp, rddir4_cache *rdc, cred_t *cr)
9155 {
9156         COMPOUND4args_clnt args;
9157         COMPOUND4res_clnt res;
9158         READDIR4args *rargs;
9159         READDIR4res_clnt *rd_res;
9160         bitmap4 rd_bitsval;
9161         nfs_argop4 argop[5];
9162         nfs_resop4 *resop;
9163         rnode4_t *rp = VTOR4(vp);
9164         mntinfo4_t *mi = VTOMI4(vp);
9165         int doqueue;
9166         u_longlong_t nodeid, pnodeid;   /* id's of dir and its parents */
```



```
9167         vnode_t *dvp;
9168         nfs_cookie4 cookie = (nfs_cookie4)rdc->nfs4_cookie;
9169         int num_ops, res_opcnt;
9170         bool_t needrecov = FALSE;
9171         nfs4_recov_state_t recov_state;
9172         hrtime_t t;
9173         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

9175         ASSERT(nfs_zone() == mi->mi_zone);
9176         ASSERT(rdc->flags & RDDIR);
9177         ASSERT(rdc->entries == NULL);

9179         /*
9180          * If rp were a stub, it should have triggered and caused
9181          * a mount for us to get this far.
9182          */
9183         ASSERT(!RP_ISSTUB(rp));

9185         num_ops = 2;
9186         if (cookie == (nfs_cookie4)0 || cookie == (nfs_cookie4)1) {
9187                 /*
9188                  * Since nfsv4 readdir may not return entries for "." and "..",
9189                  * the client must recreate them:
9190                  * To find the correct nodeid, do the following:
9191                  * For current node, get nodeid from dnlc.
9192                  * - if current node is rootvp, set pnodeid to nodeid.
9193                  * - else if parent is in the dnlc, get its nodeid from there.
9194                  * - else add LOOKUPP+GETATTR to compound.
9195                  */
9196                 nodeid = rp->r_attr.va_nodeid;
9197                 if (vp->v_flag & VROOT) {
9198                         pnodeid = nodeid;       /* root of mount point */
9199                 } else {
9200                         dvp = dnlc_lookup(vp, "..");
9201                         if (dvp != NULL && dvp != DNLC_NO_VNODE) {
9202                                 /* parent in dnlc cache - no need for otw */
9203                                 pnodeid = VTOR4(dvp)->r_attr.va_nodeid;
9204                         } else {
9205                                 /*
9206                                  * parent not in dnlc cache,
9207                                  * do lookupp to get its id
9208                                  */
9209                                 num_ops = 5;
9210                                 pnodeid = 0; /* set later by getattr parent */
9211                         }
9212                         if (dvp)
9213                                 VN_RELE(dvp);
9214                 }
9215         }
9216         recov_state.rs_flags = 0;
9217         recov_state.rs_num_retry_despite_err = 0;

9219         /* Save the original mount point security flavor */
9220         (void) save_mnt_secinfo(mi->mi_curr_serv);

9222 recov_retry:
9223         args.ctag = TAG_READDIR;

9225         args.array = argop;
9226         args.array_len = num_ops;

9228         if (e.error = nfs4_start_fop(VTOMI4(vp), vp, NULL, OH_READDIR,
9229             &recov_state, NULL)) {
9230                 /*
9231                  * If readdir a node that is a stub for a crossed mount point,
9232                  * keep the original secinfo flavor for the current file
```

```
9233                 * system, not the crossed one.
9234                 */
9235                (void) check_mnt_secinfo(mi->mi_curr_serv, vp);
9236                rdc->error = e.error;
9237                return;
9238        }

9240        /*
9241         * Determine which attrs to request for dirents.  This code
9242         * must be protected by nfs4_start/end_fop because of r_server
9243         * (which will change during failover recovery).
9244         *
9245         */
9246        if (rp->r_flags & (R4LOOKUP | R4READDIRWATTR)) {
9247                /*
9248                 * Get all vattr attrs plus filehandle and rdattr_error
9249                 */
9250                rd_bitsval = NFS4_VATTR_MASK |
9251                    FATTR4_RDATTR_ERROR_MASK |
9252                    FATTR4_FILEHANDLE_MASK;

9254                if (rp->r_flags & R4READDIRWATTR) {
9255                        mutex_enter(&rp->r_statelock);
9256                        rp->r_flags &= ~R4READDIRWATTR;
9257                        mutex_exit(&rp->r_statelock);
9258                }
9259        } else {
9260                servinfo4_t *svp = rp->r_server;

9262                /*
9263                 * Already read directory. Use readdir with
9264                 * no attrs (except for mounted_on_fileid) for updates.
9265                 */
9266                rd_bitsval = FATTR4_RDATTR_ERROR_MASK;

9268                /*
9269                 * request mounted on fileid if supported, else request
9270                 * fileid.  maybe we should verify that fileid is supported
9271                 * and request something else if not.
9272                 */
9273                (void) nfs_rw_enter_sig(&svp->sv_lock, RW_READER, 0);
9274                if (svp->sv_supp_attrs & FATTR4_MOUNTED_ON_FILEID_MASK)
9275                        rd_bitsval |= FATTR4_MOUNTED_ON_FILEID_MASK;
9276                nfs_rw_exit(&svp->sv_lock);
9277        }

9279        /* putfh directory fh */
9280        argop[0].argop = OP_CPUTFH;
9281        argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

9283        argop[1].argop = OP_READDIR;
9284        rargs = &argop[1].nfs_argop4_u.opreaddir;
9285        /*
9286         * 1 and 2 are reserved for client "." and ".." entry offset.
9287         * cookie 0 should be used over-the-wire to start reading at
9288         * the beginning of the directory excluding "." and "..".
9289         */
9290        if (rdc->nfs4_cookie == 0 ||
9291            rdc->nfs4_cookie == 1 ||
9292            rdc->nfs4_cookie == 2) {
9293                rargs->cookie = (nfs_cookie4)0;
9294                rargs->cookieverf = 0;
9295        } else {
9296                rargs->cookie = (nfs_cookie4)rdc->nfs4_cookie;
9297                mutex_enter(&rp->r_statelock);
9298                rargs->cookieverf = rp->r_cookieverf4;
```



```
9299                mutex_exit(&rp->r_statelock);
9300        }
9301        rargs->dircount = MIN(rdc->buflen, mi->mi_tsize);
9302        rargs->maxcount = mi->mi_tsize;
9303        rargs->attr_request = rd_bitsval;
9304        rargs->rdc = rdc;
9305        rargs->dvp = vp;
9306        rargs->mi = mi;
9307        rargs->cr = cr;


9310        /*
9311         * If count < than the minimum required, we return no entries
9312         * and fail with EINVAL
9313         */
9314        if (rargs->dircount < (DIRENT64_RECLEN(1) + DIRENT64_RECLEN(2))) {
9315                rdc->error = EINVAL;
9316                goto out;
9317        }

9319        if (args.array_len == 5) {
9320                /*
9321                 * Add lookupp and getattr for parent nodeid.
9322                 */
9323                argop[2].argop = OP_LOOKUPP;

9325                argop[3].argop = OP_GETFH;

9327                /* getattr parent */
9328                argop[4].argop = OP_GETATTR;
9329                argop[4].nfs_argop4_u.opgetattr.attr_request = NFS4_VATTR_MASK;
9330                argop[4].nfs_argop4_u.opgetattr.mi = mi;
9331        }

9333        doqueue = 1;

9335        if (mi->mi_io_kstats) {
9336                mutex_enter(&mi->mi_lock);
9337                kstat_runq_enter(KSTAT_IO_PTR(mi->mi_io_kstats));
9338                mutex_exit(&mi->mi_lock);
9339        }

9341        /* capture the time of this call */
9342        rargs->t = t = gethrtime();

9344        rfs4call(mi, &args, &res, cr, &doqueue, 0, &e);

9346        if (mi->mi_io_kstats) {
9347                mutex_enter(&mi->mi_lock);
9348                kstat_runq_exit(KSTAT_IO_PTR(mi->mi_io_kstats));
9349                mutex_exit(&mi->mi_lock);
9350        }

9352        needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);

9354        /*
9355         * If RPC error occurred and it isn't an error that
9356         * triggers recovery, then go ahead and fail now.
9357         */
9358        if (e.error != 0 && !needrecov) {
9359                rdc->error = e.error;
9360                goto out;
9361        }

9363        if (needrecov) {
9364                bool_t abort;
```

```
9366                NFS4_DEBUG(nfs4_client_recov_debug, (CE_NOTE,
9367                    "nfs4readdir: initiating recovery.\n"));

9369                abort = nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
9370                    NULL, OP_READDIR, NULL, NULL, NULL);
9371                if (abort == FALSE) {
9372                        nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_READDIR,
9373                            &recov_state, needrecov);
9374                        if (!e.error)
9375                                (void) xdr_free(xdr_COMPOUND4res_clnt,
9376                                    (caddr_t)&res);
9377                        if (rdc->entries != NULL) {
9378                                kmem_free(rdc->entries, rdc->entlen);
9379                                rdc->entries = NULL;
9380                        }
9381                        goto recov_retry;
9382                }

9384                if (e.error != 0) {
9385                        rdc->error = e.error;
9386                        goto out;
9387                }

9389                /* fall through for res.status case */
9390        }

9392        res_opcnt = res.array_len;

9394        /*
9395         * If compound failed first 2 ops (PUTFH+READDIR), then return
9396         * failure here.  Subsequent ops are for filling out dot-dot
9397         * dirent, and if they fail, we still want to give the caller
9398         * the dirents returned by (the successful) READDIR op, so we need
9399         * to silently ignore failure for subsequent ops (LOOKUPP+GETATTR).
9400         *
9401         * One example where PUTFH+READDIR ops would succeed but
9402         * LOOKUPP+GETATTR would fail would be a dir that has r perm
9403         * but lacks x.  In this case, a POSIX server's VOP_READDIR
9404         * would succeed; however, VOP_LOOKUP(..) would fail since no
9405         * x perm.  We need to come up with a non-vendor-specific way
9406         * for a POSIX server to return d_ino from dotdot's dirent if
9407         * client only requests mounted_on_fileid, and just say the
9408         * LOOKUPP succeeded and fill out the GETATTR.  However, if
9409         * client requested any mandatory attrs, server would be required
9410         * to fail the GETATTR op because it can't call VOP_LOOKUP+VOP_GETATTR
9411         * for dotdot.
9412         */

9414        if (res.status) {
9415                if (res_opcnt <= 2) {
9416                        e.error = geterrno4(res.status);
9417                        nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_READDIR,
9418                            &recov_state, needrecov);
9419                        nfs4_purge_stale_fh(e.error, vp, cr);
9420                        rdc->error = e.error;
9421                        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
9422                        if (rdc->entries != NULL) {
9423                                kmem_free(rdc->entries, rdc->entlen);
9424                                rdc->entries = NULL;
9425                        }
9426                        /*
9427                         * If readdir a node that is a stub for a
9428                         * crossed mount point, keep the original
9429                         * secinfo flavor for the current file system,
9430                         * not the crossed one.
```



```
9431                         */
9432                        (void) check_mnt_secinfo(mi->mi_curr_serv, vp);
9433                        return;
9434                }
9435        }

9437        resop = &res.array[1];  /* readdir res */
9438        rd_res = &resop->nfs_resop4_u.opreaddirclnt;

9440        mutex_enter(&rp->r_statelock);
9441        rp->r_cookieverf4 = rd_res->cookieverf;
9442        mutex_exit(&rp->r_statelock);

9444        /*
9445         * For "." and ".." entries
9446         * e.g.
9447         *      seek(cookie=0) -> "." entry with d_off = 1
9448         *      seek(cookie=1) -> ".." entry with d_off = 2
9449         */
9450        if (cookie == (nfs_cookie4) 0) {
9451                if (rd_res->dotp)
9452                        rd_res->dotp->d_ino = nodeid;
9453                if (rd_res->dotdotp)
9454                        rd_res->dotdotp->d_ino = pnodeid;
9455        }
9456        if (cookie == (nfs_cookie4) 1) {
9457                if (rd_res->dotdotp)
9458                        rd_res->dotdotp->d_ino = pnodeid;
9459        }


9462        /* LOOKUPP+GETATTR attemped */
9463        if (args.array_len == 5 && rd_res->dotdotp) {
9464                if (res.status == NFS4_OK && res_opcnt == 5) {
9465                        nfs_fh4 *fhp;
9466                        nfs4_sharedfh_t *sfhp;
9467                        vnode_t *pvp;
9468                        nfs4_ga_res_t *garp;

9470                        resop++;        /* lookupp */
9471                        resop++;        /* getfh   */
9472                        fhp = &resop->nfs_resop4_u.opgetfh.object;

9474                        resop++;        /* getattr of parent */

9476                        /*
9477                         * First, take care of finishing the
9478                         * readdir results.
9479                         */
9480                        garp = &resop->nfs_resop4_u.opgetattr.ga_res;
9481                        /*
9482                         * The d_ino of .. must be the inode number
9483                         * of the mounted filesystem.
9484                         */
9485                        if (garp->n4g_va.va_mask & AT_NODEID)
9486                                rd_res->dotdotp->d_ino =
9487                                    garp->n4g_va.va_nodeid;


9490                        /*
9491                         * Next, create the ".." dnlc entry
9492                         */
9493                        sfhp = sfh4_get(fhp, mi);
9494                        if (!nfs4_make_dotdot(sfhp, t, vp, cr, &pvp, 0)) {
9495                                dnlc_update(vp, "..", pvp);
9496                                VN_RELE(pvp);
```

```
9497                        }
9498                        sfh4_rele(&sfhp);
9499                }
9500        }

9502        if (mi->mi_io_kstats) {
9503                mutex_enter(&mi->mi_lock);
9504                KSTAT_IO_PTR(mi->mi_io_kstats)->reads++;
9505                KSTAT_IO_PTR(mi->mi_io_kstats)->nread += rdc->actlen;
9506                mutex_exit(&mi->mi_lock);
9507        }

9509        (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);

9511 out:
9512        /*
9513         * If readdir a node that is a stub for a crossed mount point,
9514         * keep the original secinfo flavor for the current file system,
9515         * not the crossed one.
9516         */
9517        (void) check_mnt_secinfo(mi->mi_curr_serv, vp);

9519        nfs4_end_fop(mi, vp, NULL, OH_READDIR, &recov_state, needrecov);
9520 }


9523 static int
9524 nfs4_bio(struct buf *bp, stable_how4 *stab_comm, cred_t *cr, bool_t readahead)
9525 {
9526        rnode4_t *rp = VTOR4(bp->b_vp);
9527        int count;
9528        int error;
9529        cred_t *cred_otw = NULL;
9530        offset_t offset;
9531        nfs4_open_stream_t *osp = NULL;
9532        bool_t first_time = TRUE;       /* first time getting otw cred */
9533        bool_t last_time = FALSE;       /* last time getting otw cred */

9535        ASSERT(nfs_zone() == VTOMI4(bp->b_vp)->mi_zone);

9537        DTRACE_IO1(start, struct buf *, bp);
9538        offset = ldbtob(bp->b_lblkno);

9540        if (bp->b_flags & B_READ) {
9541        read_again:
9542                /*
9543                 * Releases the osp, if it is provided.
9544                 * Puts a hold on the cred_otw and the new osp (if found).
9545                 */
9546                cred_otw = nfs4_get_otw_cred_by_osp(rp, cr, &osp,
9547                    &first_time, &last_time);
9548                error = bp->b_error = nfs4read(bp->b_vp, bp->b_un.b_addr,
9549                    offset, bp->b_bcount, &bp->b_resid, cred_otw,
9550                    readahead, NULL);
9551                crfree(cred_otw);
9552                if (!error) {
9553                        if (bp->b_resid) {
9554                                /*
9555                                 * Didn't get it all because we hit EOF,
9556                                 * zero all the memory beyond the EOF.
9557                                 */
9558                                /* bzero(rdaddr + */
9559                                bzero(bp->b_un.b_addr +
9560                                    bp->b_bcount - bp->b_resid, bp->b_resid);
9561                        }
9562                        mutex_enter(&rp->r_statelock);
```



```
9563                        if (bp->b_resid == bp->b_bcount &&
9564                            offset >= rp->r_size) {
9565                                /*
9566                                 * We didn't read anything at all as we are
9567                                 * past EOF.  Return an error indicator back
9568                                 * but don't destroy the pages (yet).
9569                                 */
9570                                error = NFS_EOF;
9571                        }
9572                        mutex_exit(&rp->r_statelock);
9573                } else if (error == EACCES && last_time == FALSE) {
9574                                goto read_again;
9575                }
9576        } else {
9577                if (!(rp->r_flags & R4STALE)) {
9578 write_again:
9579                        /*
9580                         * Releases the osp, if it is provided.
9581                         * Puts a hold on the cred_otw and the new
9582                         * osp (if found).
9583                         */
9584                        cred_otw = nfs4_get_otw_cred_by_osp(rp, cr, &osp,
9585                            &first_time, &last_time);
9586                        mutex_enter(&rp->r_statelock);
9587                        count = MIN(bp->b_bcount, rp->r_size - offset);
9588                        mutex_exit(&rp->r_statelock);
9589                        if (count < 0)
9590                                cmn_err(CE_PANIC, "nfs4_bio: write count < 0");
9591 #ifdef DEBUG
9592                        if (count == 0) {
9593                                zoneid_t zoneid = getzoneid();

9595                                zcmn_err(zoneid, CE_WARN,
9596                                    "nfs4_bio: zero length write at %lld",
9597                                    offset);
9598                                zcmn_err(zoneid, CE_CONT, "flags=0x%x, "
9599                                    "b_bcount=%ld, file size=%lld",
9600                                    rp->r_flags, (long)bp->b_bcount,
9601                                    rp->r_size);
9602                                sfh4_printfhandle(VTOR4(bp->b_vp)->r_fh);
9603                                if (nfs4_bio_do_stop)
9604                                        debug_enter("nfs4_bio");
9605                        }
9606 #endif
9607                        error = nfs4write(bp->b_vp, bp->b_un.b_addr, offset,
9608                            count, cred_otw, stab_comm);
9609                        if (error == EACCES && last_time == FALSE) {
9610                                crfree(cred_otw);
9611                                goto write_again;
9612                        }
9613                        bp->b_error = error;
9614                        if (error && error != EINTR &&
9615                            !(bp->b_vp->v_vfsp->vfs_flag & VFS_UNMOUNTED)) {
9616                                /*
9617                                 * Don't print EDQUOT errors on the console.
9618                                 * Don't print asynchronous EACCES errors.
9619                                 * Don't print EFBIG errors.
9620                                 * Print all other write errors.
9621                                 */
9622                                if (error != EDQUOT && error != EFBIG &&
9623                                    (error != EACCES ||
9624                                    !(bp->b_flags & B_ASYNC)))
9625                                        nfs4_write_error(bp->b_vp,
9626                                            error, cred_otw);
9627                                /*
9628                                 * Update r_error and r_flags as appropriate.
```

```
9629                                 * If the error was ESTALE, then mark the
9630                                 * rnode as not being writeable and save
9631                                 * the error status.  Otherwise, save any
9632                                 * errors which occur from asynchronous
9633                                 * page invalidations.  Any errors occurring
9634                                 * from other operations should be saved
9635                                 * by the caller.
9636                                 */
9637                                mutex_enter(&rp->r_statelock);
9638                                if (error == ESTALE) {
9639                                        rp->r_flags |= R4STALE;
9640                                        if (!rp->r_error)
9641                                                rp->r_error = error;
9642                                } else if (!rp->r_error &&
9643                                    (bp->b_flags &
9644                                    (B_INVAL|B_FORCE|B_ASYNC)) ==
9645                                    (B_INVAL|B_FORCE|B_ASYNC)) {
9646                                        rp->r_error = error;
9647                                }
9648                                mutex_exit(&rp->r_statelock);
9649                        }
9650                        crfree(cred_otw);
9651                } else {
9652                        error = rp->r_error;
9653                        /*
9654                         * A close may have cleared r_error, if so,
9655                         * propagate ESTALE error return properly
9656                         */
9657                        if (error == 0)
9658                                error = ESTALE;
9659                }
9660        }

9662        if (error != 0 && error != NFS_EOF)
9663                bp->b_flags |= B_ERROR;

9665        if (osp)
9666                open_stream_rele(osp, rp);

9668        DTRACE_IO1(done, struct buf *, bp);

9670        return (error);
9671 }

9673 /* ARGSUSED */
9674 int
9675 nfs4_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
9676 {
9677        return (EREMOTE);
9678 }

9680 /* ARGSUSED2 */
9681 int
9682 nfs4_rwlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
9683 {
9684        rnode4_t *rp = VTOR4(vp);

9686        if (!write_lock) {
9687                (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
9688                return (V_WRITELOCK_FALSE);
9689        }

9691        if ((rp->r_flags & R4DIRECTIO) ||
9692            (VTOMI4(vp)->mi_flags & MI4_DIRECTIO)) {
9693                (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_READER, FALSE);
9694                if (rp->r_mapcnt == 0 && !nfs4_has_pages(vp))
```



```
9695                        return (V_WRITELOCK_FALSE);
9696                nfs_rw_exit(&rp->r_rwlock);
9697        }

9699        (void) nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, FALSE);
9700        return (V_WRITELOCK_TRUE);
9701 }

9703 /* ARGSUSED */
9704 void
9705 nfs4_rwunlock(vnode_t *vp, int write_lock, caller_context_t *ctp)
9706 {
9707        rnode4_t *rp = VTOR4(vp);

9709        nfs_rw_exit(&rp->r_rwlock);
9710 }

9712 /* ARGSUSED */
9713 static int
9714 nfs4_seek(vnode_t *vp, offset_t ooff, offset_t *noffp, caller_context_t *ct)
9715 {
9716        if (nfs_zone() != VTOMI4(vp)->mi_zone)
9717                return (EIO);

9719        /*
9720         * Because we stuff the readdir cookie into the offset field
9721         * someone may attempt to do an lseek with the cookie which
9722         * we want to succeed.
9723         */
9724        if (vp->v_type == VDIR)
9725                return (0);
9726        if (*noffp < 0)
9727                return (EINVAL);
9728        return (0);
9729 }


9732 /*
9733  * Return all the pages from [off..off+len) in file
9734  */
9735 /* ARGSUSED */
9736 static int
9737 nfs4_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
9738     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
9739         enum seg_rw rw, cred_t *cr, caller_context_t *ct)
9740 {
9741        rnode4_t *rp;
9742        int error;
9743        mntinfo4_t *mi;

9745        if (nfs_zone() != VTOMI4(vp)->mi_zone)
9746                return (EIO);
9747        rp = VTOR4(vp);
9748        if (IS_SHADOW(vp, rp))
9749                vp = RTOV4(rp);

9751        if (vp->v_flag & VNOMAP)
9752                return (ENOSYS);

9754        if (protp != NULL)
9755                *protp = PROT_ALL;

9757        /*
9758         * Now validate that the caches are up to date.
9759         */
9760        if (error = nfs4_validate_caches(vp, cr))
```

```
9761                 return (error);

9763         mi = VTOMI4(vp);
9764 retry:
9765         mutex_enter(&rp->r_statelock);

9767         /*
9768          * Don't create dirty pages faster than they
9769          * can be cleaned so that the system doesn't
9770          * get imbalanced.  If the async queue is
9771          * maxed out, then wait for it to drain before
9772          * creating more dirty pages.  Also, wait for
9773          * any threads doing pagewalks in the vop_getattr
9774          * entry points so that they don't block for
9775          * long periods.
9776          */
9777         if (rw == S_CREATE) {
9778                 while ((mi->mi_max_threads != 0 &&
9779                     rp->r_awcount > 2 * mi->mi_max_threads) ||
9780                     rp->r_gcount > 0)
9781                         cv_wait(&rp->r_cv, &rp->r_statelock);
9782         }

9784         /*
9785          * If we are getting called as a side effect of an nfs_write()
9786          * operation the local file size might not be extended yet.
9787          * In this case we want to be able to return pages of zeroes.
9788          */
9789         if (off + len > rp->r_size + PAGEOFFSET && seg != segkmap) {
9790                 NFS4_DEBUG(nfs4_pageio_debug,
9791                     (CE_NOTE, "getpage beyond EOF: off=%lld, "
9792                     "len=%llu, size=%llu, attrsize =%llu", off,
9793                     (u_longlong_t)len, rp->r_size, rp->r_attr.va_size));
9794                 mutex_exit(&rp->r_statelock);
9795                 return (EFAULT);                /* beyond EOF */
9796         }

9798         mutex_exit(&rp->r_statelock);

9800         if (len <= PAGESIZE) {
9801                 error = nfs4_getapage(vp, off, len, protp, pl, plsz,
9802                     seg, addr, rw, cr);
9803                 NFS4_DEBUG(nfs4_pageio_debug && error,
9804                     (CE_NOTE, "getpage error %d; off=%lld, "
9805                     "len=%lld", error, off, (u_longlong_t)len));
9806         } else {
9807                 error = pvn_getpages(nfs4_getapage, vp, off, len, protp,
9808                     pl, plsz, seg, addr, rw, cr);
9809                 NFS4_DEBUG(nfs4_pageio_debug && error,
9810                     (CE_NOTE, "getpages error %d; off=%lld, "
9811                     "len=%lld", error, off, (u_longlong_t)len));
9812         }

9814         switch (error) {
9815         case NFS_EOF:
9816                 nfs4_purge_caches(vp, NFS4_NOPURGE_DNLC, cr, FALSE);
9817                 goto retry;
9818         case ESTALE:
9819                 nfs4_purge_stale_fh(error, vp, cr);
9820         }

9822         return (error);
9823 }

9825 /*
9826  * Called from pvn_getpages or nfs4_getpage to get a particular page.
```



```
9827  */
9828 /* ARGSUSED */
9829 static int
9830 nfs4_getapage(vnode_t *vp, u_offset_t off, size_t len, uint_t *protp,
9831     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
9832     enum seg_rw rw, cred_t *cr)
9833 {
9834         rnode4_t *rp;
9835         uint_t bsize;
9836         struct buf *bp;
9837         page_t *pp;
9838         u_offset_t lbn;
9839         u_offset_t io_off;
9840         u_offset_t blkoff;
9841         u_offset_t rablkoff;
9842         size_t io_len;
9843         uint_t blksize;
9844         int error;
9845         int readahead;
9846         int readahead_issued = 0;
9847         int ra_window; /* readahead window */
9848         page_t *pagefound;
9849         page_t *savepp;

9851         if (nfs_zone() != VTOMI4(vp)->mi_zone)
9852                 return (EIO);

9854         rp = VTOR4(vp);
9855         ASSERT(!IS_SHADOW(vp, rp));
9856         bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);

9858 reread:
9859         bp = NULL;
9860         pp = NULL;
9861         pagefound = NULL;

9863         if (pl != NULL)
9864                 pl[0] = NULL;

9866         error = 0;
9867         lbn = off / bsize;
9868         blkoff = lbn * bsize;

9870         /*
9871          * Queueing up the readahead before doing the synchronous read
9872          * results in a significant increase in read throughput because
9873          * of the increased parallelism between the async threads and
9874          * the process context.
9875          */
9876         if ((off & ((vp->v_vfsp->vfs_bsize) - 1)) == 0 &&
9877             rw != S_CREATE &&
9878             !(vp->v_flag & VNOCACHE)) {
9879                 mutex_enter(&rp->r_statelock);

9881                 /*
9882                  * Calculate the number of readaheads to do.
9883                  * a) No readaheads at offset = 0.
9884                  * b) Do maximum(nfs4_nra) readaheads when the readahead
9885                  *    window is closed.
9886                  * c) Do readaheads between 1 to (nfs4_nra - 1) depending
9887                  *    upon how far the readahead window is open or close.
9888                  * d) No readaheads if rp->r_nextr is not within the scope
9889                  *    of the readahead window (random i/o).
9890                  */

9892                 if (off == 0)
```

```
9893                    readahead = 0;
9894            else if (blkoff == rp->r_nextr)
9895                    readahead = nfs4_nra;
9896            else if (rp->r_nextr > blkoff &&
9897                ((ra_window = (rp->r_nextr - blkoff) / bsize)
9898                <= (nfs4_nra - 1)))
9899                    readahead = nfs4_nra - ra_window;
9900            else
9901                    readahead = 0;

9903            rablkoff = rp->r_nextr;
9904            while (readahead > 0 && rablkoff + bsize < rp->r_size) {
9905                    mutex_exit(&rp->r_statelock);
9906                    if (nfs4_async_readahead(vp, rablkoff + bsize,
9907                        addr + (rablkoff + bsize - off),
9908                        seg, cr, nfs4_readahead) < 0) {
9909                            mutex_enter(&rp->r_statelock);
9910                            break;
9911                    }
9912                    readahead--;
9913                    rablkoff += bsize;
9914                    /*
9915                     * Indicate that we did a readahead so
9916                     * readahead offset is not updated
9917                     * by the synchronous read below.
9918                     */
9919                    readahead_issued = 1;
9920                    mutex_enter(&rp->r_statelock);
9921                    /*
9922                     * set readahead offset to
9923                     * offset of last async readahead
9924                     * request.
9925                     */
9926                    rp->r_nextr = rablkoff;
9927            }
9928            mutex_exit(&rp->r_statelock);
9929    }

9931 again:
9932    if ((pagefound = page_exists(vp, off)) == NULL) {
9933            if (pl == NULL) {
9934                    (void) nfs4_async_readahead(vp, blkoff, addr, seg, cr,
9935                        nfs4_readahead);
9936            } else if (rw == S_CREATE) {
9937                    /*
9938                     * Block for this page is not allocated, or the offset
9939                     * is beyond the current allocation size, or we're
9940                     * allocating a swap slot and the page was not found,
9941                     * so allocate it and return a zero page.
9942                     */
9943                    if ((pp = page_create_va(vp, off,
9944                        PAGESIZE, PG_WAIT, seg, addr)) == NULL)
9945                            cmn_err(CE_PANIC, "nfs4_getapage: page_create");
9946                    io_len = PAGESIZE;
9947                    mutex_enter(&rp->r_statelock);
9948                    rp->r_nextr = off + PAGESIZE;
9949                    mutex_exit(&rp->r_statelock);
9950            } else {
9951                    /*
9952                     * Need to go to server to get a block
9953                     */
9954                    mutex_enter(&rp->r_statelock);
9955                    if (blkoff < rp->r_size &&
9956                        blkoff + bsize > rp->r_size) {
9957                            /*
9958                             * If less than a block left in
```



```
9959                            * file read less than a block.
9960                            */
9961                           if (rp->r_size <= off) {
9962                                   /*
9963                                    * Trying to access beyond EOF,
9964                                    * set up to get at least one page.
9965                                    */
9966                                   blksize = off + PAGESIZE - blkoff;
9967                           } else
9968                                   blksize = rp->r_size - blkoff;
9969                   } else if ((off == 0) ||
9970                       (off != rp->r_nextr && !readahead_issued)) {
9971                           blksize = PAGESIZE;
9972                           blkoff = off; /* block = page here */
9973                   } else
9974                           blksize = bsize;
9975                   mutex_exit(&rp->r_statelock);

9977                   pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
9978                       &io_len, blkoff, blksize, 0);

9980                   /*
9981                    * Some other thread has entered the page,
9982                    * so just use it.
9983                    */
9984                   if (pp == NULL)
9985                           goto again;

9987                   /*
9988                    * Now round the request size up to page boundaries.
9989                    * This ensures that the entire page will be
9990                    * initialized to zeroes if EOF is encountered.
9991                    */
9992                   io_len = ptob(btopr(io_len));

9994                   bp = pageio_setup(pp, io_len, vp, B_READ);
9995                   ASSERT(bp != NULL);

9997                   /*
9998                    * pageio_setup should have set b_addr to 0.  This
9999                    * is correct since we want to do I/O on a page
10000                   * boundary.  bp_mapin will use this addr to calculate
10001                   * an offset, and then set b_addr to the kernel virtual
10002                   * address it allocated for us.
10003                   */
10004                  ASSERT(bp->b_un.b_addr == 0);

10006                  bp->b_edev = 0;
10007                  bp->b_dev = 0;
10008                  bp->b_lblkno = lbtodb(io_off);
10009                  bp->b_file = vp;
10010                  bp->b_offset = (offset_t)off;
10011                  bp_mapin(bp);

10013                  /*
10014                   * If doing a write beyond what we believe is EOF,
10015                   * don't bother trying to read the pages from the
10016                   * server, we'll just zero the pages here.  We
10017                   * don't check that the rw flag is S_WRITE here
10018                   * because some implementations may attempt a
10019                   * read access to the buffer before copying data.
10020                   */
10021                  mutex_enter(&rp->r_statelock);
10022                  if (io_off >= rp->r_size && seg == segkmap) {
10023                          mutex_exit(&rp->r_statelock);
10024                          bzero(bp->b_un.b_addr, io_len);
```

```
10025                         } else {
10026                                 mutex_exit(&rp->r_statelock);
10027                                 error = nfs4_bio(bp, NULL, cr, FALSE);
10028                         }
10029
10030                         /*
10031                          * Unmap the buffer before freeing it.
10032                          */
10033                         bp_mapout(bp);
10034                         pageio_done(bp);
10035
10036                         savepp = pp;
10037                         do {
10038                                 pp->p_fsdata = C_NOCOMMIT;
10039                         } while ((pp = pp->p_next) != savepp);
10040
10041                         if (error == NFS_EOF) {
10042                                 /*
10043                                  * If doing a write system call just return
10044                                  * zeroed pages, else user tried to get pages
10045                                  * beyond EOF, return error.  We don't check
10046                                  * that the rw flag is S_WRITE here because
10047                                  * some implementations may attempt a read
10048                                  * access to the buffer before copying data.
10049                                  */
10050                                 if (seg == segkmap)
10051                                         error = 0;
10052                                 else
10053                                         error = EFAULT;
10054                         }
10055
10056                         if (!readahead_issued && !error) {
10057                                 mutex_enter(&rp->r_statelock);
10058                                 rp->r_nextr = io_off + io_len;
10059                                 mutex_exit(&rp->r_statelock);
10060                         }
10061                 }
10062         }
10063
10064 out:
10065         if (pl == NULL)
10066                 return (error);
10067
10068         if (error) {
10069                 if (pp != NULL)
10070                         pvn_read_done(pp, B_ERROR);
10071                 return (error);
10072         }
10073
10074         if (pagefound) {
10075                 se_t se = (rw == S_CREATE ? SE_EXCL : SE_SHARED);
10076
10077                 /*
10078                  * Page exists in the cache, acquire the appropriate lock.
10079                  * If this fails, start all over again.
10080                  */
10081                 if ((pp = page_lookup(vp, off, se)) == NULL) {
10082 #ifdef DEBUG
10083                         nfs4_lostpage++;
10084 #endif
10085                         goto reread;
10086                 }
10087                 pl[0] = pp;
10088                 pl[1] = NULL;
10089                 return (0);
10090         }
```



```
10092         if (pp != NULL)
10093                 pvn_plist_init(pp, pl, plsz, off, io_len, rw);
10094
10095         return (error);
10096 }
10097
10098 static void
10099 nfs4_readahead(vnode_t *vp, u_offset_t blkoff, caddr_t addr, struct seg *seg,
10100     cred_t *cr)
10101 {
10102         int error;
10103         page_t *pp;
10104         u_offset_t io_off;
10105         size_t io_len;
10106         struct buf *bp;
10107         uint_t bsize, blksize;
10108         rnode4_t *rp = VTOR4(vp);
10109         page_t *savepp;
10110
10111         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
10112
10113         bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
10114
10115         mutex_enter(&rp->r_statelock);
10116         if (blkoff < rp->r_size && blkoff + bsize > rp->r_size) {
10117                 /*
10118                  * If less than a block left in file read less
10119                  * than a block.
10120                  */
10121                 blksize = rp->r_size - blkoff;
10122         } else
10123                 blksize = bsize;
10124         mutex_exit(&rp->r_statelock);
10125
10126         pp = pvn_read_kluster(vp, blkoff, segkmap, addr,
10127             &io_off, &io_len, blkoff, blksize, 1);
10128         /*
10129          * The isra flag passed to the kluster function is 1, we may have
10130          * gotten a return value of NULL for a variety of reasons (# of free
10131          * pages < minfree, someone entered the page on the vnode etc). In all
10132          * cases, we want to punt on the readahead.
10133          */
10134         if (pp == NULL)
10135                 return;
10136
10137         /*
10138          * Now round the request size up to page boundaries.
10139          * This ensures that the entire page will be
10140          * initialized to zeroes if EOF is encountered.
10141          */
10142         io_len = ptob(btopr(io_len));
10143
10144         bp = pageio_setup(pp, io_len, vp, B_READ);
10145         ASSERT(bp != NULL);
10146
10147         /*
10148          * pageio_setup should have set b_addr to 0.  This is correct since
10149          * we want to do I/O on a page boundary. bp_mapin() will use this addr
10150          * to calculate an offset, and then set b_addr to the kernel virtual
10151          * address it allocated for us.
10152          */
10153         ASSERT(bp->b_un.b_addr == 0);
10154
10155         bp->b_edev = 0;
10156         bp->b_dev = 0;
```

```
10157         bp->b_lblkno = lbtodb(io_off);
10158         bp->b_file = vp;
10159         bp->b_offset = (offset_t)blkoff;
10160         bp_mapin(bp);

10162         /*
10163          * If doing a write beyond what we believe is EOF, don't bother trying
10164          * to read the pages from the server, we'll just zero the pages here.
10165          * We don't check that the rw flag is S_WRITE here because some
10166          * implementations may attempt a read access to the buffer before
10167          * copying data.
10168          */
10169         mutex_enter(&rp->r_statelock);
10170         if (io_off >= rp->r_size && seg == segkmap) {
10171                 mutex_exit(&rp->r_statelock);
10172                 bzero(bp->b_un.b_addr, io_len);
10173                 error = 0;
10174         } else {
10175                 mutex_exit(&rp->r_statelock);
10176                 error = nfs4_bio(bp, NULL, cr, TRUE);
10177                 if (error == NFS_EOF)
10178                         error = 0;
10179         }

10181         /*
10182          * Unmap the buffer before freeing it.
10183          */
10184         bp_mapout(bp);
10185         pageio_done(bp);

10187         savepp = pp;
10188         do {
10189                 pp->p_fsdata = C_NOCOMMIT;
10190         } while ((pp = pp->p_next) != savepp);

10192         pvn_read_done(pp, error ? B_READ | B_ERROR : B_READ);

10194         /*
10195          * In case of error set readahead offset
10196          * to the lowest offset.
10197          * pvn_read_done() calls VN_DISPOSE to destroy the pages
10198          */
10199         if (error && rp->r_nextr > io_off) {
10200                 mutex_enter(&rp->r_statelock);
10201                 if (rp->r_nextr > io_off)
10202                         rp->r_nextr = io_off;
10203                 mutex_exit(&rp->r_statelock);
10204         }
10205 }

10207 /*
10208  * Flags are composed of {B_INVAL, B_FREE, B_DONTNEED, B_FORCE}
10209  * If len == 0, do from off to EOF.
10210  *
10211  * The normal cases should be len == 0 && off == 0 (entire vp list) or
10212  * len == MAXBSIZE (from segmap_release actions), and len == PAGESIZE
10213  * (from pageout).
10214  */
10215 /* ARGSUSED */
10216 static int
10217 nfs4_putpage(vnode_t *vp, offset_t off, size_t len, int flags, cred_t *cr,
10218         caller_context_t *ct)
10219 {
10220         int error;
10221         rnode4_t *rp;
```



```
10223         ASSERT(cr != NULL);

10225         if (!(flags & B_ASYNC) && nfs_zone() != VTOMI4(vp)->mi_zone)
10226                 return (EIO);

10228         rp = VTOR4(vp);
10229         if (IS_SHADOW(vp, rp))
10230                 vp = RTOV4(rp);

10232         /*
10233          * XXX - Why should this check be made here?
10234          */
10235         if (vp->v_flag & VNOMAP)
10236                 return (ENOSYS);

10238         if (len == 0 && !(flags & B_INVAL) &&
10239             (vp->v_vfsp->vfs_flag & VFS_RDONLY))
10240                 return (0);

10242         mutex_enter(&rp->r_statelock);
10243         rp->r_count++;
10244         mutex_exit(&rp->r_statelock);
10245         error = nfs4_putpages(vp, off, len, flags, cr);
10246         mutex_enter(&rp->r_statelock);
10247         rp->r_count--;
10248         cv_broadcast(&rp->r_cv);
10249         mutex_exit(&rp->r_statelock);

10251         return (error);
10252 }

10254 /*
10255  * Write out a single page, possibly klustering adjacent dirty pages.
10256  */
10257 int
10258 nfs4_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp, size_t *lenp,
10259     int flags, cred_t *cr)
10260 {
10261         u_offset_t io_off;
10262         u_offset_t lbn_off;
10263         u_offset_t lbn;
10264         size_t io_len;
10265         uint_t bsize;
10266         int error;
10267         rnode4_t *rp;

10269         ASSERT(!(vp->v_vfsp->vfs_flag & VFS_RDONLY));
10270         ASSERT(pp != NULL);
10271         ASSERT(cr != NULL);
10272         ASSERT((flags & B_ASYNC) || nfs_zone() == VTOMI4(vp)->mi_zone);

10274         rp = VTOR4(vp);
10275         ASSERT(rp->r_count > 0);
10276         ASSERT(!IS_SHADOW(vp, rp));

10278         bsize = MAX(vp->v_vfsp->vfs_bsize, PAGESIZE);
10279         lbn = pp->p_offset / bsize;
10280         lbn_off = lbn * bsize;

10282         /*
10283          * Find a kluster that fits in one block, or in
10284          * one page if pages are bigger than blocks.  If
10285          * there is less file space allocated than a whole
10286          * page, we'll shorten the i/o request below.
10287          */
10288         pp = pvn_write_kluster(vp, pp, &io_off, &io_len, lbn_off,
```

```
10289             roundup(bsize, PAGESIZE), flags);
10290
10291         /*
10292          * pvn_write_kluster shouldn't have returned a page with offset
10293          * behind the original page we were given.  Verify that.
10294          */
10295         ASSERT((pp->p_offset / bsize) >= lbn);
10296
10297         /*
10298          * Now pp will have the list of kept dirty pages marked for
10299          * write back.  It will also handle invalidation and freeing
10300          * of pages that are not dirty.  Check for page length rounding
10301          * problems.
10302          */
10303         if (io_off + io_len > lbn_off + bsize) {
10304                 ASSERT((io_off + io_len) - (lbn_off + bsize) < PAGESIZE);
10305                 io_len = lbn_off + bsize - io_off;
10306         }
10307         /*
10308          * The R4MODINPROGRESS flag makes sure that nfs4_bio() sees a
10309          * consistent value of r_size. R4MODINPROGRESS is set in writerp4().
10310          * When R4MODINPROGRESS is set it indicates that a uiomove() is in
10311          * progress and the r_size has not been made consistent with the
10312          * new size of the file. When the uiomove() completes the r_size is
10313          * updated and the R4MODINPROGRESS flag is cleared.
10314          *
10315          * The R4MODINPROGRESS flag makes sure that nfs4_bio() sees a
10316          * consistent value of r_size. Without this handshaking, it is
10317          * possible that nfs4_bio() picks  up the old value of r_size
10318          * before the uiomove() in writerp4() completes. This will result
10319          * in the write through nfs4_bio() being dropped.
10320          *
10321          * More precisely, there is a window between the time the uiomove()
10322          * completes and the time the r_size is updated. If a VOP_PUTPAGE()
10323          * operation intervenes in this window, the page will be picked up,
10324          * because it is dirty (it will be unlocked, unless it was
10325          * pagecreate'd). When the page is picked up as dirty, the dirty
10326          * bit is reset (pvn_getdirty()). In nfs4write(), r_size is
10327          * checked. This will still be the old size. Therefore the page will
10328          * not be written out. When segmap_release() calls VOP_PUTPAGE(),
10329          * the page will be found to be clean and the write will be dropped.
10330          */
10331         if (rp->r_flags & R4MODINPROGRESS) {
10332                 mutex_enter(&rp->r_statelock);
10333                 if ((rp->r_flags & R4MODINPROGRESS) &&
10334                     rp->r_modaddr + MAXBSIZE > io_off &&
10335                     rp->r_modaddr < io_off + io_len) {
10336                         page_t *plist;
10337                         /*
10338                          * A write is in progress for this region of the file.
10339                          * If we did not detect R4MODINPROGRESS here then this
10340                          * path through nfs_putapage() would eventually go to
10341                          * nfs4_bio() and may not write out all of the data
10342                          * in the pages. We end up losing data. So we decide
10343                          * to set the modified bit on each page in the page
10344                          * list and mark the rnode with R4DIRTY. This write
10345                          * will be restarted at some later time.
10346                          */
10347                         plist = pp;
10348                         while (plist != NULL) {
10349                                 pp = plist;
10350                                 page_sub(&plist, pp);
10351                                 hat_setmod(pp);
10352                                 page_io_unlock(pp);
10353                                 page_unlock(pp);
10354                         }
```



```
10355                         rp->r_flags |= R4DIRTY;
10356                         mutex_exit(&rp->r_statelock);
10357                         if (offp)
10358                                 *offp = io_off;
10359                         if (lenp)
10360                                 *lenp = io_len;
10361                         return (0);
10362                 }
10363                 mutex_exit(&rp->r_statelock);
10364         }
10365
10366         if (flags & B_ASYNC) {
10367                 error = nfs4_async_putapage(vp, pp, io_off, io_len, flags, cr,
10368                     nfs4_sync_putapage);
10369         } else
10370                 error = nfs4_sync_putapage(vp, pp, io_off, io_len, flags, cr);
10371
10372         if (offp)
10373                 *offp = io_off;
10374         if (lenp)
10375                 *lenp = io_len;
10376         return (error);
10377 }
10378
10379 static int
10380 nfs4_sync_putapage(vnode_t *vp, page_t *pp, u_offset_t io_off, size_t io_len,
10381     int flags, cred_t *cr)
10382 {
10383         int error;
10384         rnode4_t *rp;
10385
10386         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
10387
10388         flags |= B_WRITE;
10389
10390         error = nfs4_rdwrlbn(vp, pp, io_off, io_len, flags, cr);
10391
10392         rp = VTOR4(vp);
10393
10394         if ((error == ENOSPC || error == EDQUOT || error == EFBIG ||
10395             error == EACCES) &&
10396             (flags & (B_INVAL|B_FORCE)) != (B_INVAL|B_FORCE)) {
10397                 if (!(rp->r_flags & R4OUTOFSPACE)) {
10398                         mutex_enter(&rp->r_statelock);
10399                         rp->r_flags |= R4OUTOFSPACE;
10400                         mutex_exit(&rp->r_statelock);
10401                 }
10402                 flags |= B_ERROR;
10403                 pvn_write_done(pp, flags);
10404                 /*
10405                  * If this was not an async thread, then try again to
10406                  * write out the pages, but this time, also destroy
10407                  * them whether or not the write is successful.  This
10408                  * will prevent memory from filling up with these
10409                  * pages and destroying them is the only alternative
10410                  * if they can't be written out.
10411                  *
10412                  * Don't do this if this is an async thread because
10413                  * when the pages are unlocked in pvn_write_done,
10414                  * some other thread could have come along, locked
10415                  * them, and queued for an async thread.  It would be
10416                  * possible for all of the async threads to be tied
10417                  * up waiting to lock the pages again and they would
10418                  * all already be locked and waiting for an async
10419                  * thread to handle them.  Deadlock.
10420                  */
```

```
10421                 if (!(flags & B_ASYNC)) {
10422                         error = nfs4_putpage(vp, io_off, io_len,
10423                             B_INVAL | B_FORCE, cr, NULL);
10424                 }
10425         } else {
10426                 if (error)
10427                         flags |= B_ERROR;
10428                 else if (rp->r_flags & R4OUTOFSPACE) {
10429                         mutex_enter(&rp->r_statelock);
10430                         rp->r_flags &= ~R4OUTOFSPACE;
10431                         mutex_exit(&rp->r_statelock);
10432                 }
10433                 pvn_write_done(pp, flags);
10434                 if (freemem < desfree)
10435                         (void) nfs4_commit_vp(vp, (u_offset_t)0, 0, cr,
10436                             NFS4_WRITE_NOWAIT);
10437         }

10439         return (error);
10440 }

10442 #ifdef DEBUG
10443 int nfs4_force_open_before_mmap = 0;
10444 #endif

10446 /* ARGSUSED */
10447 static int
10448 nfs4_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
10449     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
10450     caller_context_t *ct)
10451 {
10452         struct segvn_crargs vn_a;
10453         int error = 0;
10454         rnode4_t *rp = VTOR4(vp);
10455         mntinfo4_t *mi = VTOMI4(vp);

10457         if (nfs_zone() != VTOMI4(vp)->mi_zone)
10458                 return (EIO);

10460         if (vp->v_flag & VNOMAP)
10461                 return (ENOSYS);

10463         if (off < 0 || (off + len) < 0)
10464                 return (ENXIO);

10466         if (vp->v_type != VREG)
10467                 return (ENODEV);

10469         /*
10470          * If the file is delegated to the client don't do anything.
10471          * If the file is not delegated, then validate the data cache.
10472          */
10473         mutex_enter(&rp->r_statev4_lock);
10474         if (rp->r_deleg_type == OPEN_DELEGATE_NONE) {
10475                 mutex_exit(&rp->r_statev4_lock);
10476                 error = nfs4_validate_caches(vp, cr);
10477                 if (error)
10478                         return (error);
10479         } else {
10480                 mutex_exit(&rp->r_statev4_lock);
10481         }

10483         /*
10484          * Check to see if the vnode is currently marked as not cachable.
10485          * This means portions of the file are locked (through VOP_FRLOCK).
10486          * In this case the map request must be refused.  We use
```



```
10487          * rp->r_lkserlock to avoid a race with concurrent lock requests.
10488          *
10489          * Atomically increment r_inmap after acquiring r_rwlock. The
10490          * idea here is to acquire r_rwlock to block read/write and
10491          * not to protect r_inmap. r_inmap will inform nfs4_read/write()
10492          * that we are in nfs4_map(). Now, r_rwlock is acquired in order
10493          * and we can prevent the deadlock that would have occurred
10494          * when nfs4_addmap() would have acquired it out of order.
10495          *
10496          * Since we are not protecting r_inmap by any lock, we do not
10497          * hold any lock when we decrement it. We atomically decrement
10498          * r_inmap after we release r_lkserlock.
10499          */

10501         if (nfs_rw_enter_sig(&rp->r_rwlock, RW_WRITER, INTR4(vp)))
10502                 return (EINTR);
10503         atomic_add_int(&rp->r_inmap, 1);
10504         nfs_rw_exit(&rp->r_rwlock);

10506         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_READER, INTR4(vp))) {
10507                 atomic_add_int(&rp->r_inmap, -1);
10508                 return (EINTR);
10509         }


10512         if (vp->v_flag & VNOCACHE) {
10513                 error = EAGAIN;
10514                 goto done;
10515         }

10517         /*
10518          * Don't allow concurrent locks and mapping if mandatory locking is
10519          * enabled.
10520          */
10521         if (flk_has_remote_locks(vp)) {
10522                 struct vattr va;
10523                 va.va_mask = AT_MODE;
10524                 error = nfs4getattr(vp, &va, cr);
10525                 if (error != 0)
10526                         goto done;
10527                 if (MANDLOCK(vp, va.va_mode)) {
10528                         error = EAGAIN;
10529                         goto done;
10530                 }
10531         }

10533         /*
10534          * It is possible that the rnode has a lost lock request that we
10535          * are still trying to recover, and that the request conflicts with
10536          * this map request.
10537          *
10538          * An alternative approach would be for nfs4_safemap() to consider
10539          * queued lock requests when deciding whether to set or clear
10540          * VNOCACHE.  This would require the frlock code path to call
10541          * nfs4_safemap() after enqueing a lost request.
10542          */
10543         if (nfs4_map_lost_lock_conflict(vp)) {
10544                 error = EAGAIN;
10545                 goto done;
10546         }

10548         as_rangelock(as);
10549         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
10550         if (error != 0) {
10551                 as_rangeunlock(as);
10552                 goto done;
```

```
10553         }

10555         if (vp->v_type == VREG) {
10556                 /*
10557                  * We need to retrieve the open stream
10558                  */
10559                 nfs4_open_stream_t      *osp = NULL;
10560                 nfs4_open_owner_t       *oop = NULL;

10562                 oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
10563                 if (oop != NULL) {
10564                         /* returns with 'os_sync_lock' held */
10565                         osp = find_open_stream(oop, rp);
10566                         open_owner_rele(oop);
10567                 }
10568                 if (osp == NULL) {
10569 #ifdef DEBUG
10570                         if (nfs4_force_open_before_mmap) {
10571                                 error = EIO;
10572                                 goto done;
10573                         }
10574 #endif
10575                         /* returns with 'os_sync_lock' held */
10576                         error = open_and_get_osp(vp, cr, &osp);
10577                         if (osp == NULL) {
10578                                 NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE,
10579                                     "nfs4_map: we tried to OPEN the file "
10580                                     "but again no osp, so fail with EIO"));
10581                                 goto done;
10582                         }
10583                 }

10585                 if (osp->os_failed_reopen) {
10586                         mutex_exit(&osp->os_sync_lock);
10587                         open_stream_rele(osp, rp);
10588                         NFS4_DEBUG(nfs4_open_stream_debug, (CE_NOTE,
10589                             "nfs4_map: os_failed_reopen set on "
10590                             "osp %p, cr %p, rp %s", (void *)osp,
10591                             (void *)cr, rnode4info(rp)));
10592                         error = EIO;
10593                         goto done;
10594                 }
10595                 mutex_exit(&osp->os_sync_lock);
10596                 open_stream_rele(osp, rp);
10597         }

10599         vn_a.vp = vp;
10600         vn_a.offset = off;
10601         vn_a.type = (flags & MAP_TYPE);
10602         vn_a.prot = (uchar_t)prot;
10603         vn_a.maxprot = (uchar_t)maxprot;
10604         vn_a.flags = (flags & ~MAP_TYPE);
10605         vn_a.cred = cr;
10606         vn_a.amp = NULL;
10607         vn_a.szc = 0;
10608         vn_a.lgrp_mem_policy_flags = 0;

10610         error = as_map(as, *addrp, len, segvn_create, &vn_a);
10611         as_rangeunlock(as);

10613 done:
10614         nfs_rw_exit(&rp->r_lkserlock);
10615         atomic_add_int(&rp->r_inmap, -1);
10616         return (error);
10617 }
```



```
10619 /*
10620  * We're most likely dealing with a kernel module that likes to READ
10621  * and mmap without OPENing the file (ie: lookup/read/mmap), so lets
10622  * officially OPEN the file to create the necessary client state
10623  * for bookkeeping of os_mmap_read/write counts.
10624  *
10625  * Since VOP_MAP only passes in a pointer to the vnode rather than
10626  * a double pointer, we can't handle the case where nfs4open_otw()
10627  * returns a different vnode than the one passed into VOP_MAP (since
10628  * VOP_DELMAP will not see the vnode nfs4open_otw used).  In this case,
10629  * we return NULL and let nfs4_map() fail.  Note: the only case where
10630  * this should happen is if the file got removed and replaced with the
10631  * same name on the server (in addition to the fact that we're trying
10632  * to VOP_MAP withouth VOP_OPENing the file in the first place).
10633  */
10634 static int
10635 open_and_get_osp(vnode_t *map_vp, cred_t *cr, nfs4_open_stream_t **ospp)
10636 {
10637         rnode4_t                *rp, *drp;
10638         vnode_t                 *dvp, *open_vp;
10639         char                    file_name[MAXNAMELEN];
10640         int                     just_created;
10641         nfs4_open_stream_t      *osp;
10642         nfs4_open_owner_t       *oop;
10643         int                     error;

10645         *ospp = NULL;
10646         open_vp = map_vp;

10648         rp = VTOR4(open_vp);
10649         if ((error = vtodv(open_vp, &dvp, cr, TRUE)) != 0)
10650                 return (error);
10651         drp = VTOR4(dvp);

10653         if (nfs_rw_enter_sig(&drp->r_rwlock, RW_READER, INTR4(dvp))) {
10654                 VN_RELE(dvp);
10655                 return (EINTR);
10656         }

10658         if ((error = vtoname(open_vp, file_name, MAXNAMELEN)) != 0) {
10659                 nfs_rw_exit(&drp->r_rwlock);
10660                 VN_RELE(dvp);
10661                 return (error);
10662         }

10664         mutex_enter(&rp->r_statev4_lock);
10665         if (rp->created_v4) {
10666                 rp->created_v4 = 0;
10667                 mutex_exit(&rp->r_statev4_lock);

10669                 dnlc_update(dvp, file_name, open_vp);
10670                 /* This is needed so we don't bump the open ref count */
10671                 just_created = 1;
10672         } else {
10673                 mutex_exit(&rp->r_statev4_lock);
10674                 just_created = 0;
10675         }

10677         VN_HOLD(map_vp);

10679         error = nfs4open_otw(dvp, file_name, NULL, &open_vp, cr, 0, FREAD, 0,
10680             just_created);
10681         if (error) {
10682                 nfs_rw_exit(&drp->r_rwlock);
10683                 VN_RELE(dvp);
10684                 VN_RELE(map_vp);
```

```
10685                   return (error);
10686           }

10688           nfs_rw_exit(&drp->r_rwlock);
10689           VN_RELE(dvp);

10691           /*
10692            * If nfs4open_otw() returned a different vnode then "undo"
10693            * the open and return failure to the caller.
10694            */
10695           if (!VN_CMP(open_vp, map_vp)) {
10696                   nfs4_error_t e;

10698                   NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE, "open_and_get_osp: "
10699                       "open returned a different vnode"));
10700                   /*
10701                    * If there's an error, ignore it,
10702                    * and let VOP_INACTIVE handle it.
10703                    */
10704                   (void) nfs4close_one(open_vp, NULL, cr, FREAD, NULL, &e,
10705                       CLOSE_NORM, 0, 0, 0);
10706                   VN_RELE(map_vp);
10707                   return (EIO);
10708           }

10710           VN_RELE(map_vp);

10712           oop = find_open_owner(cr, NFS4_PERM_CREATED, VTOMI4(open_vp));
10713           if (!oop) {
10714                   nfs4_error_t e;

10716                   NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE, "open_and_get_osp: "
10717                       "no open owner"));
10718                   /*
10719                    * If there's an error, ignore it,
10720                    * and let VOP_INACTIVE handle it.
10721                    */
10722                   (void) nfs4close_one(open_vp, NULL, cr, FREAD, NULL, &e,
10723                       CLOSE_NORM, 0, 0, 0);
10724                   return (EIO);
10725           }
10726           osp = find_open_stream(oop, rp);
10727           open_owner_rele(oop);
10728           *ospp = osp;
10729           return (0);
10730   }

10732   /*
10733    * Please be aware that when this function is called, the address space write
10734    * a_lock is held.  Do not put over the wire calls in this function.
10735    */
10736   /* ARGSUSED */
10737   static int
10738   nfs4_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
10739       size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
10740       caller_context_t *ct)
10741   {
10742           rnode4_t                *rp;
10743           int                     error = 0;
10744           mntinfo4_t              *mi;

10746           mi = VTOMI4(vp);
10747           rp = VTOR4(vp);

10749           if (nfs_zone() != mi->mi_zone)
10750                   return (EIO);
```



```
10751           if (vp->v_flag & VNOMAP)
10752                   return (ENOSYS);

10754           /*
10755            * Don't need to update the open stream first, since this
10756            * mmap can't add any additional share access that isn't
10757            * already contained in the open stream (for the case where we
10758            * open/mmap/only update rp->r_mapcnt/server reboots/reopen doesn't
10759            * take into account os_mmap_read[write] counts).
10760            */
10761           atomic_add_long((ulong_t *)&rp->r_mapcnt, btopr(len));

10763           if (vp->v_type == VREG) {
10764                   /*
10765                    * We need to retrieve the open stream and update the counts.
10766                    * If there is no open stream here, something is wrong.
10767                    */
10768                   nfs4_open_stream_t      *osp = NULL;
10769                   nfs4_open_owner_t       *oop = NULL;

10771                   oop = find_open_owner(cr, NFS4_PERM_CREATED, mi);
10772                   if (oop != NULL) {
10773                           /* returns with 'os_sync_lock' held */
10774                           osp = find_open_stream(oop, rp);
10775                           open_owner_rele(oop);
10776                   }
10777                   if (osp == NULL) {
10778                           NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE,
10779                               "nfs4_addmap: we should have an osp"
10780                               "but we don't, so fail with EIO"));
10781                           error = EIO;
10782                           goto out;
10783                   }

10785                   NFS4_DEBUG(nfs4_mmap_debug, (CE_NOTE, "nfs4_addmap: osp %p,"
10786                       " pages %ld, prot 0x%x", (void *)osp, btopr(len), prot));

10788                   /*
10789                    * Update the map count in the open stream.
10790                    * This is necessary in the case where we
10791                    * open/mmap/close/, then the server reboots, and we
10792                    * attempt to reopen.  If the mmap doesn't add share
10793                    * access then we send an invalid reopen with
10794                    * access = NONE.
10795                    *
10796                    * We need to specifically check each PROT_* so a mmap
10797                    * call of (PROT_WRITE | PROT_EXEC) will ensure us both
10798                    * read and write access.  A simple comparison of prot
10799                    * to ~PROT_WRITE to determine read access is insufficient
10800                    * since prot can be |= with PROT_USER, etc.
10801                    */

10803                   /*
10804                    * Unless we're MAP_SHARED, no sense in adding os_mmap_write
10805                    */
10806                   if ((flags & MAP_SHARED) && (maxprot & PROT_WRITE))
10807                           osp->os_mmap_write += btopr(len);
10808                   if (maxprot & PROT_READ)
10809                           osp->os_mmap_read += btopr(len);
10810                   if (maxprot & PROT_EXEC)
10811                           osp->os_mmap_read += btopr(len);
10812                   /*
10813                    * Ensure that os_mmap_read gets incremented, even if
10814                    * maxprot were to look like PROT_NONE.
10815                    */
10816                   if (!(maxprot & PROT_READ) && !(maxprot & PROT_WRITE) &&
```

```
10817                     !(maxprot & PROT_EXEC))
10818                         osp->os_mmap_read += btopr(len);
10819                 osp->os_mapcnt += btopr(len);
10820                 mutex_exit(&osp->os_sync_lock);
10821                 open_stream_rele(osp, rp);
10822         }

10824 out:
10825         /*
10826          * If we got an error, then undo our
10827          * incrementing of 'r_mapcnt'.
10828          */

10830         if (error) {
10831                 atomic_add_long((ulong_t *)&rp->r_mapcnt, -btopr(len));
10832                 ASSERT(rp->r_mapcnt >= 0);
10833         }
10834         return (error);
10835 }

10837 /* ARGSUSED */
10838 static int
10839 nfs4_cmp(vnode_t *vp1, vnode_t *vp2, caller_context_t *ct)
10840 {

10842         return (VTOR4(vp1) == VTOR4(vp2));
10843 }

10845 /* ARGSUSED */
10846 static int
10847 nfs4_frlock(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
10848     offset_t offset, struct flk_callback *flk_cbp, cred_t *cr,
10849     caller_context_t *ct)
10850 {
10851         int rc;
10852         u_offset_t start, end;
10853         rnode4_t *rp;
10854         int error = 0, intr = INTR4(vp);
10855         nfs4_error_t e;

10857         if (nfs_zone() != VTOMI4(vp)->mi_zone)
10858                 return (EIO);

10860         /* check for valid cmd parameter */
10861         if (cmd != F_GETLK && cmd != F_SETLK && cmd != F_SETLKW)
10862                 return (EINVAL);

10864         /* Verify l_type. */
10865         switch (bfp->l_type) {
10866         case F_RDLCK:
10867                 if (cmd != F_GETLK && !(flag & FREAD))
10868                         return (EBADF);
10869                 break;
10870         case F_WRLCK:
10871                 if (cmd != F_GETLK && !(flag & FWRITE))
10872                         return (EBADF);
10873                 break;
10874         case F_UNLCK:
10875                 intr = 0;
10876                 break;

10878         default:
10879                 return (EINVAL);
10880         }

10882         /* check the validity of the lock range */
```



```
10883         if (rc = flk_convert_lock_data(vp, bfp, &start, &end, offset))
10884                 return (rc);
10885         if (rc = flk_check_lock_data(start, end, MAXEND))
10886                 return (rc);

10888         /*
10889          * If the filesystem is mounted using local locking, pass the
10890          * request off to the local locking code.
10891          */
10892         if (VTOMI4(vp)->mi_flags & MI4_LLOCK || vp->v_type != VREG) {
10893                 if (cmd == F_SETLK || cmd == F_SETLKW) {
10894                         /*
10895                          * For complete safety, we should be holding
10896                          * r_lkserlock.  However, we can't call
10897                          * nfs4_safelock and then fs_frlock while
10898                          * holding r_lkserlock, so just invoke
10899                          * nfs4_safelock and expect that this will
10900                          * catch enough of the cases.
10901                          */
10902                         if (!nfs4_safelock(vp, bfp, cr))
10903                                 return (EAGAIN);
10904                 }
10905                 return (fs_frlock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
10906         }

10908         rp = VTOR4(vp);

10910         /*
10911          * Check whether the given lock request can proceed, given the
10912          * current file mappings.
10913          */
10914         if (nfs_rw_enter_sig(&rp->r_lkserlock, RW_WRITER, intr))
10915                 return (EINTR);
10916         if (cmd == F_SETLK || cmd == F_SETLKW) {
10917                 if (!nfs4_safelock(vp, bfp, cr)) {
10918                         rc = EAGAIN;
10919                         goto done;
10920                 }
10921         }

10923         /*
10924          * Flush the cache after waiting for async I/O to finish.  For new
10925          * locks, this is so that the process gets the latest bits from the
10926          * server.  For unlocks, this is so that other clients see the
10927          * latest bits once the file has been unlocked.  If currently dirty
10928          * pages can't be flushed, then don't allow a lock to be set.  But
10929          * allow unlocks to succeed, to avoid having orphan locks on the
10930          * server.
10931          */
10932         if (cmd != F_GETLK) {
10933                 mutex_enter(&rp->r_statelock);
10934                 while (rp->r_count > 0) {
10935                         if (intr) {
10936                                 klwp_t *lwp = ttolwp(curthread);

10938                                 if (lwp != NULL)
10939                                         lwp->lwp_nostop++;
10940                                 if (cv_wait_sig(&rp->r_cv,
10941                                     &rp->r_statelock) == 0) {
10942                                         if (lwp != NULL)
10943                                                 lwp->lwp_nostop--;
10944                                         rc = EINTR;
10945                                         break;
10946                                 }
10947                                 if (lwp != NULL)
10948                                         lwp->lwp_nostop--;
```

```
10949                                   } else
10950                                           cv_wait(&rp->r_cv, &rp->r_statelock);
10951                           }
10952                           mutex_exit(&rp->r_statelock);
10953                           if (rc != 0)
10954                                   goto done;
10955                           error = nfs4_putpage(vp, (offset_t)0, 0, B_INVAL, cr, ct);
10956                           if (error) {
10957                                   if (error == ENOSPC || error == EDQUOT) {
10958                                           mutex_enter(&rp->r_statelock);
10959                                           if (!rp->r_error)
10960                                                   rp->r_error = error;
10961                                           mutex_exit(&rp->r_statelock);
10962                                   }
10963                                   if (bfp->l_type != F_UNLCK) {
10964                                           rc = ENOLCK;
10965                                           goto done;
10966                                   }
10967                           }
10968                   }

10970                   /*
10971                    * Call the lock manager to do the real work of contacting
10972                    * the server and obtaining the lock.
10973                    */
10974                   nfs4frlock(NFS4_LCK_CTYPE_NORM, vp, cmd, bfp, flag, offset,
10975                       cr, &e, NULL, NULL);
10976                   rc = e.error;

10978                   if (rc == 0)
10979                           nfs4_lockcompletion(vp, cmd);

10981           done:
10982                   nfs_rw_exit(&rp->r_lkserlock);

10984                   return (rc);
10985           }

10987           /*
10988            * Free storage space associated with the specified vnode.  The portion
10989            * to be freed is specified by bfp->l_start and bfp->l_len (already
10990            * normalized to a "whence" of 0).
10991            *
10992            * This is an experimental facility whose continued existence is not
10993            * guaranteed.  Currently, we only support the special case
10994            * of l_len == 0, meaning free to end of file.
10995            */
10996           /* ARGSUSED */
10997           static int
10998           nfs4_space(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
10999               offset_t offset, cred_t *cr, caller_context_t *ct)
11000           {
11001                   int error;

11003                   if (nfs_zone() != VTOMI4(vp)->mi_zone)
11004                           return (EIO);
11005                   ASSERT(vp->v_type == VREG);
11006                   if (cmd != F_FREESP)
11007                           return (EINVAL);

11009                   error = convoff(vp, bfp, 0, offset);
11010                   if (!error) {
11011                           ASSERT(bfp->l_start >= 0);
11012                           if (bfp->l_len == 0) {
11013                                   struct vattr va;
```



```
11015                                   va.va_mask = AT_SIZE;
11016                                   va.va_size = bfp->l_start;
11017                                   error = nfs4setattr(vp, &va, 0, cr, NULL);

11019                                   if (error == 0 && bfp->l_start == 0)
11020                                           vnevent_truncate(vp, ct);
11021                           } else
11022                                   error = EINVAL;
11023                   }

11025                   return (error);
11026           }

11028           /* ARGSUSED */
11029           int
11030           nfs4_realvp(vnode_t *vp, vnode_t **vpp, caller_context_t *ct)
11031           {
11032                   rnode4_t *rp;
11033                   rp = VTOR4(vp);

11035                   if (vp->v_type == VREG && IS_SHADOW(vp, rp)) {
11036                           vp = RTOV4(rp);
11037                   }
11038                   *vpp = vp;
11039                   return (0);
11040           }

11042           /*
11043            * Setup and add an address space callback to do the work of the delmap call.
11044            * The callback will (and must be) deleted in the actual callback function.
11045            *
11046            * This is done in order to take care of the problem that we have with holding
11047            * the address space's a_lock for a long period of time (e.g. if the NFS server
11048            * is down).  Callbacks will be executed in the address space code while the
11049            * a_lock is not held.  Holding the address space's a_lock causes things such
11050            * as ps and fork to hang because they are trying to acquire this lock as well.
11051            */
11052           /* ARGSUSED */
11053           static int
11054           nfs4_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
11055               size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
11056               caller_context_t *ct)
11057           {
11058                   int                     caller_found;
11059                   int                     error;
11060                   rnode4_t                *rp;
11061                   nfs4_delmap_args_t      *dmapp;
11062                   nfs4_delmapcall_t       *delmap_call;

11064                   if (vp->v_flag & VNOMAP)
11065                           return (ENOSYS);

11067                   /*
11068                    * A process may not change zones if it has NFS pages mmap'ed
11069                    * in, so we can't legitimately get here from the wrong zone.
11070                    */
11071                   ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11073                   rp = VTOR4(vp);

11075                   /*
11076                    * The way that the address space of this process deletes its mapping
11077                    * of this file is via the following call chains:
11078                    * - as_free()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11079                    * - as_unmap()->SEGOP_UNMAP()/segvn_unmap()->VOP_DELMAP()/nfs4_delmap()
11080                    *
```

```
11081          * With the use of address space callbacks we are allowed to drop the
11082          * address space lock, a_lock, while executing the NFS operations that
11083          * need to go over the wire.  Returning EAGAIN to the caller of this
11084          * function is what drives the execution of the callback that we add
11085          * below.  The callback will be executed by the address space code
11086          * after dropping the a_lock.  When the callback is finished, since
11087          * we dropped the a_lock, it must be re-acquired and segvn_unmap()
11088          * is called again on the same segment to finish the rest of the work
11089          * that needs to happen during unmapping.
11090          *
11091          * This action of calling back into the segment driver causes
11092          * nfs4_delmap() to get called again, but since the callback was
11093          * already executed at this point, it already did the work and there
11094          * is nothing left for us to do.
11095          *
11096          * To Summarize:
11097          * - The first time nfs4_delmap is called by the current thread is when
11098          * we add the caller associated with this delmap to the delmap caller
11099          * list, add the callback, and return EAGAIN.
11100          * - The second time in this call chain when nfs4_delmap is called we
11101          * will find this caller in the delmap caller list and realize there
11102          * is no more work to do thus removing this caller from the list and
11103          * returning the error that was set in the callback execution.
11104          */
11105         caller_found = nfs4_find_and_delete_delmapcall(rp, &error);
11106         if (caller_found) {
11107                 /*
11108                  * 'error' is from the actual delmap operations.  To avoid
11109                  * hangs, we need to handle the return of EAGAIN differently
11110                  * since this is what drives the callback execution.
11111                  * In this case, we don't want to return EAGAIN and do the
11112                  * callback execution because there are none to execute.
11113                  */
11114                 if (error == EAGAIN)
11115                         return (0);
11116                 else
11117                         return (error);
11118         }
11119
11120         /* current caller was not in the list */
11121         delmap_call = nfs4_init_delmapcall();
11122
11123         mutex_enter(&rp->r_statelock);
11124         list_insert_tail(&rp->r_indelmap, delmap_call);
11125         mutex_exit(&rp->r_statelock);
11126
11127         dmapp = kmem_alloc(sizeof (nfs4_delmap_args_t), KM_SLEEP);
11128
11129         dmapp->vp = vp;
11130         dmapp->off = off;
11131         dmapp->addr = addr;
11132         dmapp->len = len;
11133         dmapp->prot = prot;
11134         dmapp->maxprot = maxprot;
11135         dmapp->flags = flags;
11136         dmapp->cr = cr;
11137         dmapp->caller = delmap_call;
11138
11139         error = as_add_callback(as, nfs4_delmap_callback, dmapp,
11140             AS_UNMAP_EVENT, addr, len, KM_SLEEP);
11141
11142         return (error ? error : EAGAIN);
11143 }
11144
11145 static nfs4_delmapcall_t *
11146 nfs4_init_delmapcall()
```



```
11147 {
11148         nfs4_delmapcall_t       *delmap_call;
11149
11150         delmap_call = kmem_alloc(sizeof (nfs4_delmapcall_t), KM_SLEEP);
11151         delmap_call->call_id = curthread;
11152         delmap_call->error = 0;
11153
11154         return (delmap_call);
11155 }
11156
11157 static void
11158 nfs4_free_delmapcall(nfs4_delmapcall_t *delmap_call)
11159 {
11160         kmem_free(delmap_call, sizeof (nfs4_delmapcall_t));
11161 }
11162
11163 /*
11164  * Searches for the current delmap caller (based on curthread) in the list of
11165  * callers.  If it is found, we remove it and free the delmap caller.
11166  * Returns:
11167  *      0 if the caller wasn't found
11168  *      1 if the caller was found, removed and freed.  *errp will be set
11169  *      to what the result of the delmap was.
11170  */
11171 static int
11172 nfs4_find_and_delete_delmapcall(rnode4_t *rp, int *errp)
11173 {
11174         nfs4_delmapcall_t       *delmap_call;
11175
11176         /*
11177          * If the list doesn't exist yet, we create it and return
11178          * that the caller wasn't found.  No list = no callers.
11179          */
11180         mutex_enter(&rp->r_statelock);
11181         if (!(rp->r_flags & R4DELMAPLIST)) {
11182                 /* The list does not exist */
11183                 list_create(&rp->r_indelmap, sizeof (nfs4_delmapcall_t),
11184                     offsetof(nfs4_delmapcall_t, call_node));
11185                 rp->r_flags |= R4DELMAPLIST;
11186                 mutex_exit(&rp->r_statelock);
11187                 return (0);
11188         } else {
11189                 /* The list exists so search it */
11190                 for (delmap_call = list_head(&rp->r_indelmap);
11191                     delmap_call != NULL;
11192                     delmap_call = list_next(&rp->r_indelmap, delmap_call)) {
11193                         if (delmap_call->call_id == curthread) {
11194                                 /* current caller is in the list */
11195                                 *errp = delmap_call->error;
11196                                 list_remove(&rp->r_indelmap, delmap_call);
11197                                 mutex_exit(&rp->r_statelock);
11198                                 nfs4_free_delmapcall(delmap_call);
11199                                 return (1);
11200                         }
11201                 }
11202         }
11203         mutex_exit(&rp->r_statelock);
11204         return (0);
11205 }
11206
11207 /*
11208  * Remove some pages from an mmap'd vnode.  Just update the
11209  * count of pages.  If doing close-to-open, then flush and
11210  * commit all of the pages associated with this file.
11211  * Otherwise, start an asynchronous page flush to write out
11212  * any dirty pages.  This will also associate a credential
```

```
 * with the rnode which can be used to write the pages.
 */
/* ARGSUSED */
static void
nfs4_delmap_callback(struct as *as, void *arg, uint_t event)
{
        nfs4_error_t            e = { 0, NFS4_OK, RPC_SUCCESS };
        rnode4_t                *rp;
        mntinfo4_t              *mi;
        nfs4_delmap_args_t      *dmapp = (nfs4_delmap_args_t *)arg;

        rp = VTOR4(dmapp->vp);
        mi = VTOMI4(dmapp->vp);

        atomic_add_long((ulong_t *)&rp->r_mapcnt, -btopr(dmapp->len));
        ASSERT(rp->r_mapcnt >= 0);

        /*
         * Initiate a page flush and potential commit if there are
         * pages, the file system was not mounted readonly, the segment
         * was mapped shared, and the pages themselves were writeable.
         */
        if (nfs4_has_pages(dmapp->vp) &&
            !(dmapp->vp->v_vfsp->vfs_flag & VFS_RDONLY) &&
            dmapp->flags == MAP_SHARED && (dmapp->maxprot & PROT_WRITE)) {
                mutex_enter(&rp->r_statelock);
                rp->r_flags |= R4DIRTY;
                mutex_exit(&rp->r_statelock);
                e.error = nfs4_putpage_commit(dmapp->vp, dmapp->off,
                    dmapp->len, dmapp->cr);
                if (!e.error) {
                        mutex_enter(&rp->r_statelock);
                        e.error = rp->r_error;
                        rp->r_error = 0;
                        mutex_exit(&rp->r_statelock);
                }
        } else
                e.error = 0;

        if ((rp->r_flags & R4DIRECTIO) || (mi->mi_flags & MI4_DIRECTIO))
                (void) nfs4_putpage(dmapp->vp, dmapp->off, dmapp->len,
                    B_INVAL, dmapp->cr, NULL);

        if (e.error) {
                e.stat = puterrno4(e.error);
                nfs4_queue_fact(RF_DELMAP_CB_ERR, mi, e.stat, 0,
                    OP_COMMIT, FALSE, NULL, 0, dmapp->vp);
                dmapp->caller->error = e.error;
        }

        /* Check to see if we need to close the file */

        if (dmapp->vp->v_type == VREG) {
                nfs4close_one(dmapp->vp, NULL, dmapp->cr, 0, NULL, &e,
                    CLOSE_DELMAP, dmapp->len, dmapp->maxprot, dmapp->flags);

                if (e.error != 0 || e.stat != NFS4_OK) {
                        /*
                         * Since it is possible that e.error == 0 and
                         * e.stat != NFS4_OK (and vice versa),
                         * we do the proper checking in order to get both
                         * e.error and e.stat reporting the correct info.
                         */
                        if (e.stat == NFS4_OK)
                                e.stat = puterrno4(e.error);
                        if (e.error == 0)
```



```
                                e.error = geterrno4(e.stat);

                        nfs4_queue_fact(RF_DELMAP_CB_ERR, mi, e.stat, 0,
                            OP_CLOSE, FALSE, NULL, 0, dmapp->vp);
                        dmapp->caller->error = e.error;
                }
        }

        (void) as_delete_callback(as, arg);
        kmem_free(dmapp, sizeof (nfs4_delmap_args_t));
}


static uint_t
fattr4_maxfilesize_to_bits(uint64_t ll)
{
        uint_t l = 1;

        if (ll == 0) {
                return (0);
        }

        if (ll & 0xffffffff00000000) {
                l += 32; ll >>= 32;
        }
        if (ll & 0xffff0000) {
                l += 16; ll >>= 16;
        }
        if (ll & 0xff00) {
                l += 8; ll >>= 8;
        }
        if (ll & 0xf0) {
                l += 4; ll >>= 4;
        }
        if (ll & 0xc) {
                l += 2; ll >>= 2;
        }
        if (ll & 0x2) {
                l += 1;
        }
        return (l);
}

static int
nfs4_have_xattrs(vnode_t *vp, ulong_t *valp, cred_t *cr)
{
        vnode_t *avp = NULL;
        int error;

        if ((error = nfs4lookup_xattr(vp, "", &avp,
            LOOKUP_XATTR, cr)) == 0)
                error = do_xattr_exists_check(avp, valp, cr);
        if (avp)
                VN_RELE(avp);

        return (error);
}

/* ARGSUSED */
int
nfs4_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
        caller_context_t *ct)
{
        int error;
        hrtime_t t;
        rnode4_t *rp;
```

```
11345         nfs4_ga_res_t gar;
11346         nfs4_ga_ext_res_t ger;

11348         gar.n4g_ext_res = &ger;

11350         if (nfs_zone() != VTOMI4(vp)->mi_zone)
11351                 return (EIO);
11352         if (cmd == _PC_PATH_MAX || cmd == _PC_SYMLINK_MAX) {
11353                 *valp = MAXPATHLEN;
11354                 return (0);
11355         }
11356         if (cmd == _PC_ACL_ENABLED) {
11357                 *valp = _ACL_ACE_ENABLED;
11358                 return (0);
11359         }

11361         rp = VTOR4(vp);
11362         if (cmd == _PC_XATTR_EXISTS) {
11363                 /*
11364                  * The existence of the xattr directory is not sufficient
11365                  * for determining whether generic user attributes exists.
11366                  * The attribute directory could only be a transient directory
11367                  * used for Solaris sysattr support.  Do a small readdir
11368                  * to verify if the only entries are sysattrs or not.
11369                  *
11370                  * pc4_xattr_valid can be only be trusted when r_xattr_dir
11371                  * is NULL.  Once the xadir vp exists, we can create xattrs,
11372                  * and we don't have any way to update the "base" object's
11373                  * pc4_xattr_exists from the xattr or xadir.  Maybe FEM
11374                  * could help out.
11375                  */
11376                 if (ATTRCACHE4_VALID(vp) && rp->r_pathconf.pc4_xattr_valid &&
11377                     rp->r_xattr_dir == NULL) {
11378                         return (nfs4_have_xattrs(vp, valp, cr));
11379                 }
11380         } else {  /* OLD CODE */
11381                 if (ATTRCACHE4_VALID(vp)) {
11382                         mutex_enter(&rp->r_statelock);
11383                         if (rp->r_pathconf.pc4_cache_valid) {
11384                                 error = 0;
11385                                 switch (cmd) {
11386                                 case _PC_FILESIZEBITS:
11387                                         *valp =
11388                                             rp->r_pathconf.pc4_filesizebits;
11389                                         break;
11390                                 case _PC_LINK_MAX:
11391                                         *valp =
11392                                             rp->r_pathconf.pc4_link_max;
11393                                         break;
11394                                 case _PC_NAME_MAX:
11395                                         *valp =
11396                                             rp->r_pathconf.pc4_name_max;
11397                                         break;
11398                                 case _PC_CHOWN_RESTRICTED:
11399                                         *valp =
11400                                             rp->r_pathconf.pc4_chown_restricted;
11401                                         break;
11402                                 case _PC_NO_TRUNC:
11403                                         *valp =
11404                                             rp->r_pathconf.pc4_no_trunc;
11405                                         break;
11406                                 default:
11407                                         error = EINVAL;
11408                                         break;
11409                                 }
11410                                 mutex_exit(&rp->r_statelock);
```



```
11411 #ifdef DEBUG
11412                                 nfs4_pathconf_cache_hits++;
11413 #endif
11414                                 return (error);
11415                         }
11416                         mutex_exit(&rp->r_statelock);
11417                 }
11418         }
11419 #ifdef DEBUG
11420         nfs4_pathconf_cache_misses++;
11421 #endif

11423         t = gethrtime();

11425         error = nfs4_attr_otw(vp, TAG_PATHCONF, &gar, NFS4_PATHCONF_MASK, cr);

11427         if (error) {
11428                 mutex_enter(&rp->r_statelock);
11429                 rp->r_pathconf.pc4_cache_valid = FALSE;
11430                 rp->r_pathconf.pc4_xattr_valid = FALSE;
11431                 mutex_exit(&rp->r_statelock);
11432                 return (error);
11433         }

11435         /* interpret the max filesize */
11436         gar.n4g_ext_res->n4g_pc4.pc4_filesizebits =
11437             fattr4_maxfilesize_to_bits(gar.n4g_ext_res->n4g_maxfilesize);

11439         /* Store the attributes we just received */
11440         nfs4_attr_cache(vp, &gar, t, cr, TRUE, NULL);

11442         switch (cmd) {
11443         case _PC_FILESIZEBITS:
11444                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_filesizebits;
11445                 break;
11446         case _PC_LINK_MAX:
11447                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_link_max;
11448                 break;
11449         case _PC_NAME_MAX:
11450                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_name_max;
11451                 break;
11452         case _PC_CHOWN_RESTRICTED:
11453                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_chown_restricted;
11454                 break;
11455         case _PC_NO_TRUNC:
11456                 *valp = gar.n4g_ext_res->n4g_pc4.pc4_no_trunc;
11457                 break;
11458         case _PC_XATTR_EXISTS:
11459                 if (gar.n4g_ext_res->n4g_pc4.pc4_xattr_exists) {
11460                         if (error = nfs4_have_xattrs(vp, valp, cr))
11461                                 return (error);
11462                 }
11463                 break;
11464         default:
11465                 return (EINVAL);
11466         }

11468         return (0);
11469 }

11471 /*
11472  * Called by async thread to do synchronous pageio. Do the i/o, wait
11473  * for it to complete, and cleanup the page list when done.
11474  */
11475 static int
11476 nfs4_sync_pageio(vnode_t *vp, page_t *pp, u_offset_t io_off, size_t io_len,
```

```
11477     int flags, cred_t *cr)
11478 {
11479         int error;

11481         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11483         error = nfs4_rdwrlbn(vp, pp, io_off, io_len, flags, cr);
11484         if (flags & B_READ)
11485                 pvn_read_done(pp, (error ? B_ERROR : 0) | flags);
11486         else
11487                 pvn_write_done(pp, (error ? B_ERROR : 0) | flags);
11488         return (error);
11489 }

11491 /* ARGSUSED */
11492 static int
11493 nfs4_pageio(vnode_t *vp, page_t *pp, u_offset_t io_off, size_t io_len,
11494         int flags, cred_t *cr, caller_context_t *ct)
11495 {
11496         int error;
11497         rnode4_t *rp;

11499         if (!(flags & B_ASYNC) && nfs_zone() != VTOMI4(vp)->mi_zone)
11500                 return (EIO);

11502         if (pp == NULL)
11503                 return (EINVAL);

11505         rp = VTOR4(vp);
11506         mutex_enter(&rp->r_statelock);
11507         rp->r_count++;
11508         mutex_exit(&rp->r_statelock);

11510         if (flags & B_ASYNC) {
11511                 error = nfs4_async_pageio(vp, pp, io_off, io_len, flags, cr,
11512                     nfs4_sync_pageio);
11513         } else
11514                 error = nfs4_rdwrlbn(vp, pp, io_off, io_len, flags, cr);
11515         mutex_enter(&rp->r_statelock);
11516         rp->r_count--;
11517         cv_broadcast(&rp->r_cv);
11518         mutex_exit(&rp->r_statelock);
11519         return (error);
11520 }

11522 /* ARGSUSED */
11523 static void
11524 nfs4_dispose(vnode_t *vp, page_t *pp, int fl, int dn, cred_t *cr,
11525         caller_context_t *ct)
11526 {
11527         int error;
11528         rnode4_t *rp;
11529         page_t *plist;
11530         page_t *pptr;
11531         offset3 offset;
11532         count3 len;
11533         k_sigset_t smask;

11535         /*
11536          * We should get called with fl equal to either B_FREE or
11537          * B_INVAL.  Any other value is illegal.
11538          *
11539          * The page that we are either supposed to free or destroy
11540          * should be exclusive locked and its io lock should not
11541          * be held.
11542          */
```



```
11543         ASSERT(fl == B_FREE || fl == B_INVAL);
11544         ASSERT((PAGE_EXCL(pp) && !page_iolock_assert(pp)) || panicstr);

11546         rp = VTOR4(vp);

11548         /*
11549          * If the page doesn't need to be committed or we shouldn't
11550          * even bother attempting to commit it, then just make sure
11551          * that the p_fsdata byte is clear and then either free or
11552          * destroy the page as appropriate.
11553          */
11554         if (pp->p_fsdata == C_NOCOMMIT || (rp->r_flags & R4STALE)) {
11555                 pp->p_fsdata = C_NOCOMMIT;
11556                 if (fl == B_FREE)
11557                         page_free(pp, dn);
11558                 else
11559                         page_destroy(pp, dn);
11560                 return;
11561         }

11563         /*
11564          * If there is a page invalidation operation going on, then
11565          * if this is one of the pages being destroyed, then just
11566          * clear the p_fsdata byte and then either free or destroy
11567          * the page as appropriate.
11568          */
11569         mutex_enter(&rp->r_statelock);
11570         if ((rp->r_flags & R4TRUNCATE) && pp->p_offset >= rp->r_truncaddr) {
11571                 mutex_exit(&rp->r_statelock);
11572                 pp->p_fsdata = C_NOCOMMIT;
11573                 if (fl == B_FREE)
11574                         page_free(pp, dn);
11575                 else
11576                         page_destroy(pp, dn);
11577                 return;
11578         }

11580         /*
11581          * If we are freeing this page and someone else is already
11582          * waiting to do a commit, then just unlock the page and
11583          * return.  That other thread will take care of commiting
11584          * this page.  The page can be freed sometime after the
11585          * commit has finished.  Otherwise, if the page is marked
11586          * as delay commit, then we may be getting called from
11587          * pvn_write_done, one page at a time.   This could result
11588          * in one commit per page, so we end up doing lots of small
11589          * commits instead of fewer larger commits.  This is bad,
11590          * we want do as few commits as possible.
11591          */
11592         if (fl == B_FREE) {
11593                 if (rp->r_flags & R4COMMITWAIT) {
11594                         page_unlock(pp);
11595                         mutex_exit(&rp->r_statelock);
11596                         return;
11597                 }
11598                 if (pp->p_fsdata == C_DELAYCOMMIT) {
11599                         pp->p_fsdata = C_COMMIT;
11600                         page_unlock(pp);
11601                         mutex_exit(&rp->r_statelock);
11602                         return;
11603                 }
11604         }

11606         /*
11607          * Check to see if there is a signal which would prevent an
11608          * attempt to commit the pages from being successful.  If so,
```

```
11609          * then don't bother with all of the work to gather pages and
11610          * generate the unsuccessful RPC.  Just return from here and
11611          * let the page be committed at some later time.
11612          */
11613         sigintr(&smask, VTOMI4(vp)->mi_flags & MI4_INT);
11614         if (ttolwp(curthread) != NULL && ISSIG(curthread, JUSTLOOKING)) {
11615                 sigunintr(&smask);
11616                 page_unlock(pp);
11617                 mutex_exit(&rp->r_statelock);
11618                 return;
11619         }
11620         sigunintr(&smask);

11622         /*
11623          * We are starting to need to commit pages, so let's try
11624          * to commit as many as possible at once to reduce the
11625          * overhead.
11626          *
11627          * Set the 'commit inprogress' state bit.  We must
11628          * first wait until any current one finishes.  Then
11629          * we initialize the c_pages list with this page.
11630          */
11631         while (rp->r_flags & R4COMMIT) {
11632                 rp->r_flags |= R4COMMITWAIT;
11633                 cv_wait(&rp->r_commit.c_cv, &rp->r_statelock);
11634                 rp->r_flags &= ~R4COMMITWAIT;
11635         }
11636         rp->r_flags |= R4COMMIT;
11637         mutex_exit(&rp->r_statelock);
11638         ASSERT(rp->r_commit.c_pages == NULL);
11639         rp->r_commit.c_pages = pp;
11640         rp->r_commit.c_commbase = (offset3)pp->p_offset;
11641         rp->r_commit.c_commlen = PAGESIZE;

11643         /*
11644          * Gather together all other pages which can be committed.
11645          * They will all be chained off r_commit.c_pages.
11646          */
11647         nfs4_get_commit(vp);

11649         /*
11650          * Clear the 'commit inprogress' status and disconnect
11651          * the list of pages to be committed from the rnode.
11652          * At this same time, we also save the starting offset
11653          * and length of data to be committed on the server.
11654          */
11655         plist = rp->r_commit.c_pages;
11656         rp->r_commit.c_pages = NULL;
11657         offset = rp->r_commit.c_commbase;
11658         len = rp->r_commit.c_commlen;
11659         mutex_enter(&rp->r_statelock);
11660         rp->r_flags &= ~R4COMMIT;
11661         cv_broadcast(&rp->r_commit.c_cv);
11662         mutex_exit(&rp->r_statelock);

11664         if (curproc == proc_pageout || curproc == proc_fsflush ||
11665             nfs_zone() != VTOMI4(vp)->mi_zone) {
11666                 nfs4_async_commit(vp, plist, offset, len,
11667                     cr, do_nfs4_async_commit);
11668                 return;
11669         }

11671         /*
11672          * Actually generate the COMMIT op over the wire operation.
11673          */
11674         error = nfs4_commit(vp, (offset4)offset, (count4)len, cr);
```



```
11676         /*
11677          * If we got an error during the commit, just unlock all
11678          * of the pages.  The pages will get retransmitted to the
11679          * server during a putpage operation.
11680          */
11681         if (error) {
11682                 while (plist != NULL) {
11683                         pptr = plist;
11684                         page_sub(&plist, pptr);
11685                         page_unlock(pptr);
11686                 }
11687                 return;
11688         }

11690         /*
11691          * We've tried as hard as we can to commit the data to stable
11692          * storage on the server.  We just unlock the rest of the pages
11693          * and clear the commit required state.  They will be put
11694          * onto the tail of the cachelist if they are nolonger
11695          * mapped.
11696          */
11697         while (plist != pp) {
11698                 pptr = plist;
11699                 page_sub(&plist, pptr);
11700                 pptr->p_fsdata = C_NOCOMMIT;
11701                 page_unlock(pptr);
11702         }

11704         /*
11705          * It is possible that nfs4_commit didn't return error but
11706          * some other thread has modified the page we are going
11707          * to free/destroy.
11708          *    In this case we need to rewrite the page. Do an explicit check
11709          * before attempting to free/destroy the page. If modified, needs to
11710          * be rewritten so unlock the page and return.
11711          */
11712         if (hat_ismod(pp)) {
11713                 pp->p_fsdata = C_NOCOMMIT;
11714                 page_unlock(pp);
11715                 return;
11716         }

11718         /*
11719          * Now, as appropriate, either free or destroy the page
11720          * that we were called with.
11721          */
11722         pp->p_fsdata = C_NOCOMMIT;
11723         if (fl == B_FREE)
11724                 page_free(pp, dn);
11725         else
11726                 page_destroy(pp, dn);
11727 }

11729 /*
11730  * Commit requires that the current fh be the file written to.
11731  * The compound op structure is:
11732  *      PUTFH(file), COMMIT
11733  */
11734 static int
11735 nfs4_commit(vnode_t *vp, offset4 offset, count4 count, cred_t *cr)
11736 {
11737         COMPOUND4args_clnt args;
11738         COMPOUND4res_clnt res;
11739         COMMIT4res *cm_res;
11740         nfs_argop4 argop[2];
```

```
11741         nfs_resop4 *resop;
11742         int doqueue;
11743         mntinfo4_t *mi;
11744         rnode4_t *rp;
11745         cred_t *cred_otw = NULL;
11746         bool_t needrecov = FALSE;
11747         nfs4_recov_state_t recov_state;
11748         nfs4_open_stream_t *osp = NULL;
11749         bool_t first_time = TRUE;       /* first time getting OTW cred */
11750         bool_t last_time = FALSE;       /* last time getting OTW cred */
11751         nfs4_error_t e = { 0, NFS4_OK, RPC_SUCCESS };

11753         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11755         rp = VTOR4(vp);

11757         mi = VTOMI4(vp);
11758         recov_state.rs_flags = 0;
11759         recov_state.rs_num_retry_despite_err = 0;
11760 get_commit_cred:
11761         /*
11762          * Releases the osp, if a valid open stream is provided.
11763          * Puts a hold on the cred_otw and the new osp (if found).
11764          */
11765         cred_otw = nfs4_get_otw_cred_by_osp(rp, cr, &osp,
11766             &first_time, &last_time);
11767         args.ctag = TAG_COMMIT;
11768 recov_retry:
11769         /*
11770          * Commit ops: putfh file; commit
11771          */
11772         args.array_len = 2;
11773         args.array = argop;

11775         e.error = nfs4_start_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11776             &recov_state, NULL);
11777         if (e.error) {
11778                 crfree(cred_otw);
11779                 if (osp != NULL)
11780                         open_stream_rele(osp, rp);
11781                 return (e.error);
11782         }

11784         /* putfh directory */
11785         argop[0].argop = OP_CPUTFH;
11786         argop[0].nfs_argop4_u.opcputfh.sfh = rp->r_fh;

11788         /* commit */
11789         argop[1].argop = OP_COMMIT;
11790         argop[1].nfs_argop4_u.opcommit.offset = offset;
11791         argop[1].nfs_argop4_u.opcommit.count = count;

11793         doqueue = 1;
11794         rfs4call(mi, &args, &res, cred_otw, &doqueue, 0, &e);

11796         needrecov = nfs4_needs_recovery(&e, FALSE, mi->mi_vfsp);
11797         if (!needrecov && e.error) {
11798                 nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT, &recov_state,
11799                     needrecov);
11800                 crfree(cred_otw);
11801                 if (e.error == EACCES && last_time == FALSE)
11802                         goto get_commit_cred;
11803                 if (osp != NULL)
11804                         open_stream_rele(osp, rp);
11805                 return (e.error);
11806         }
```



```
11808         if (needrecov) {
11809                 if (nfs4_start_recovery(&e, VTOMI4(vp), vp, NULL, NULL,
11810                     NULL, OP_COMMIT, NULL, NULL, NULL) == FALSE) {
11811                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11812                             &recov_state, needrecov);
11813                         if (!e.error)
11814                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
11815                                     (caddr_t)&res);
11816                         goto recov_retry;
11817                 }
11818                 if (e.error) {
11819                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11820                             &recov_state, needrecov);
11821                         crfree(cred_otw);
11822                         if (osp != NULL)
11823                                 open_stream_rele(osp, rp);
11824                         return (e.error);
11825                 }
11826                 /* fall through for res.status case */
11827         }

11829         if (res.status) {
11830                 e.error = geterrno4(res.status);
11831                 if (e.error == EACCES && last_time == FALSE) {
11832                         crfree(cred_otw);
11833                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11834                             &recov_state, needrecov);
11835                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
11836                         goto get_commit_cred;
11837                 }
11838                 /*
11839                  * Can't do a nfs4_purge_stale_fh here because this
11840                  * can cause a deadlock.  nfs4_commit can
11841                  * be called from nfs4_dispose which can be called
11842                  * indirectly via pvn_vplist_dirty.  nfs4_purge_stale_fh
11843                  * can call back to pvn_vplist_dirty.
11844                  */
11845                 if (e.error == ESTALE) {
11846                         mutex_enter(&rp->r_statelock);
11847                         rp->r_flags |= R4STALE;
11848                         if (!rp->r_error)
11849                                 rp->r_error = e.error;
11850                         mutex_exit(&rp->r_statelock);
11851                         PURGE_ATTRCACHE4(vp);
11852                 } else {
11853                         mutex_enter(&rp->r_statelock);
11854                         if (!rp->r_error)
11855                                 rp->r_error = e.error;
11856                         mutex_exit(&rp->r_statelock);
11857                 }
11858         } else {
11859                 ASSERT(rp->r_flags & R4HAVEVERF);
11860                 resop = &res.array[1];  /* commit res */
11861                 cm_res = &resop->nfs_resop4_u.opcommit;
11862                 mutex_enter(&rp->r_statelock);
11863                 if (cm_res->writeverf == rp->r_writeverf) {
11864                         mutex_exit(&rp->r_statelock);
11865                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
11866                         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT,
11867                             &recov_state, needrecov);
11868                         crfree(cred_otw);
11869                         if (osp != NULL)
11870                                 open_stream_rele(osp, rp);
11871                         return (0);
11872                 }
```

```
11873                 nfs4_set_mod(vp);
11874                 rp->r_writeverf = cm_res->writeverf;
11875                 mutex_exit(&rp->r_statelock);
11876                 e.error = NFS_VERF_MISMATCH;
11877         }

11879         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
11880         nfs4_end_fop(VTOMI4(vp), vp, NULL, OH_COMMIT, &recov_state, needrecov);
11881         crfree(cred_otw);
11882         if (osp != NULL)
11883                 open_stream_rele(osp, rp);

11885         return (e.error);
11886 }

11888 static void
11889 nfs4_set_mod(vnode_t *vp)
11890 {
11891         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

11893         /* make sure we're looking at the master vnode, not a shadow */
11894         pvn_vplist_setdirty(RTOV4(VTOR4(vp)), nfs_setmod_check);
11895 }

11897 /*
11898  * This function is used to gather a page list of the pages which
11899  * can be committed on the server.
11900  *
11901  * The calling thread must have set R4COMMIT.  This bit is used to
11902  * serialize access to the commit structure in the rnode.  As long
11903  * as the thread has set R4COMMIT, then it can manipulate the commit
11904  * structure without requiring any other locks.
11905  *
11906  * When this function is called from nfs4_dispose() the page passed
11907  * into nfs4_dispose() will be SE_EXCL locked, and so this function
11908  * will skip it. This is not a problem since we initially add the
11909  * page to the r_commit page list.
11910  *
11911  */
11912 static void
11913 nfs4_get_commit(vnode_t *vp)
11914 {
11915         rnode4_t *rp;
11916         page_t *pp;
11917         kmutex_t *vphm;

11919         rp = VTOR4(vp);

11921         ASSERT(rp->r_flags & R4COMMIT);

11923         /* make sure we're looking at the master vnode, not a shadow */

11925         if (IS_SHADOW(vp, rp))
11926                 vp = RTOV4(rp);

11928         vphm = page_vnode_mutex(vp);
11929         mutex_enter(vphm);

11931         /*
11932          * If there are no pages associated with this vnode, then
11933          * just return.
11934          */
11935         if ((pp = vp->v_pages) == NULL) {
11936                 mutex_exit(vphm);
11937                 return;
11938         }
```



```
11940         /*
11941          * Step through all of the pages associated with this vnode
11942          * looking for pages which need to be committed.
11943          */
11944         do {
11945                 /* Skip marker pages. */
11946                 if (pp->p_hash == PVN_VPLIST_HASH_TAG)
11947                         continue;

11949                 /*
11950                  * First short-cut everything (without the page_lock)
11951                  * and see if this page does not need to be committed
11952                  * or is modified if so then we'll just skip it.
11953                  */
11954                 if (pp->p_fsdata == C_NOCOMMIT || hat_ismod(pp))
11955                         continue;

11957                 /*
11958                  * Attempt to lock the page.  If we can't, then
11959                  * someone else is messing with it or we have been
11960                  * called from nfs4_dispose and this is the page that
11961                  * nfs4_dispose was called with.. anyway just skip it.
11962                  */
11963                 if (!page_trylock(pp, SE_EXCL))
11964                         continue;

11966                 /*
11967                  * Lets check again now that we have the page lock.
11968                  */
11969                 if (pp->p_fsdata == C_NOCOMMIT || hat_ismod(pp)) {
11970                         page_unlock(pp);
11971                         continue;
11972                 }

11974                 /* this had better not be a free page */
11975                 ASSERT(PP_ISFREE(pp) == 0);

11977                 /*
11978                  * The page needs to be committed and we locked it.
11979                  * Update the base and length parameters and add it
11980                  * to r_pages.
11981                  */
11982                 if (rp->r_commit.c_pages == NULL) {
11983                         rp->r_commit.c_commbase = (offset3)pp->p_offset;
11984                         rp->r_commit.c_commlen = PAGESIZE;
11985                 } else if (pp->p_offset < rp->r_commit.c_commbase) {
11986                         rp->r_commit.c_commlen = rp->r_commit.c_commbase -
11987                             (offset3)pp->p_offset + rp->r_commit.c_commlen;
11988                         rp->r_commit.c_commbase = (offset3)pp->p_offset;
11989                 } else if ((rp->r_commit.c_commbase + rp->r_commit.c_commlen)
11990                     <= pp->p_offset) {
11991                         rp->r_commit.c_commlen = (offset3)pp->p_offset -
11992                             rp->r_commit.c_commbase + PAGESIZE;
11993                 }
11994                 page_add(&rp->r_commit.c_pages, pp);
11995         } while ((pp = pp->p_vpnext) != vp->v_pages);

11997         mutex_exit(vphm);
11998 }

12000 /*
12001  * This routine is used to gather together a page list of the pages
12002  * which are to be committed on the server.  This routine must not
12003  * be called if the calling thread holds any locked pages.
12004  *
```

```
12005  * The calling thread must have set R4COMMIT.  This bit is used to
12006  * serialize access to the commit structure in the rnode.  As long
12007  * as the thread has set R4COMMIT, then it can manipulate the commit
12008  * structure without requiring any other locks.
12009  */
12010 static void
12011 nfs4_get_commit_range(vnode_t *vp, u_offset_t soff, size_t len)
12012 {

12014         rnode4_t *rp;
12015         page_t *pp;
12016         u_offset_t end;
12017         u_offset_t off;
12018         ASSERT(len != 0);
12019         rp = VTOR4(vp);
12020         ASSERT(rp->r_flags & R4COMMIT);

12022         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12024         /* make sure we're looking at the master vnode, not a shadow */

12026         if (IS_SHADOW(vp, rp))
12027                 vp = RTOV4(rp);

12029         /*
12030          * If there are no pages associated with this vnode, then
12031          * just return.
12032          */
12033         if ((pp = vp->v_pages) == NULL)
12034                 return;
12035         /*
12036          * Calculate the ending offset.
12037          */
12038         end = soff + len;
12039         for (off = soff; off < end; off += PAGESIZE) {
12040                 /*
12041                  * Lookup each page by vp, offset.
12042                  */
12043                 if ((pp = page_lookup_nowait(vp, off, SE_EXCL)) == NULL)
12044                         continue;
12045                 /*
12046                  * If this page does not need to be committed or is
12047                  * modified, then just skip it.
12048                  */
12049                 if (pp->p_fsdata == C_NOCOMMIT || hat_ismod(pp)) {
12050                         page_unlock(pp);
12051                         continue;
12052                 }

12054                 ASSERT(PP_ISFREE(pp) == 0);
12055                 /*
12056                  * The page needs to be committed and we locked it.
12057                  * Update the base and length parameters and add it
12058                  * to r_pages.
12059                  */
12060                 if (rp->r_commit.c_pages == NULL) {
12061                         rp->r_commit.c_commbase = (offset3)pp->p_offset;
12062                         rp->r_commit.c_commlen = PAGESIZE;
12063                 } else {
12064                         rp->r_commit.c_commlen = (offset3)pp->p_offset -
12065                             rp->r_commit.c_commbase + PAGESIZE;
12066                 }
12067                 page_add(&rp->r_commit.c_pages, pp);
12068         }
12069 }
```



```
12071 /*
12072  * Called from nfs4_close(), nfs4_fsync() and nfs4_delmap().
12073  * Flushes and commits data to the server.
12074  */
12075 static int
12076 nfs4_putpage_commit(vnode_t *vp, offset_t poff, size_t plen, cred_t *cr)
12077 {
12078         int error;
12079         verifier4 write_verf;
12080         rnode4_t *rp = VTOR4(vp);

12082         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12084         /*
12085          * Flush the data portion of the file and then commit any
12086          * portions which need to be committed.  This may need to
12087          * be done twice if the server has changed state since
12088          * data was last written.  The data will need to be
12089          * rewritten to the server and then a new commit done.
12090          *
12091          * In fact, this may need to be done several times if the
12092          * server is having problems and crashing while we are
12093          * attempting to do this.
12094          */

12096 top:
12097         /*
12098          * Do a flush based on the poff and plen arguments.  This
12099          * will synchronously write out any modified pages in the
12100          * range specified by (poff, plen). This starts all of the
12101          * i/o operations which will be waited for in the next
12102          * call to nfs4_putpage
12103          */

12105         mutex_enter(&rp->r_statelock);
12106         write_verf = rp->r_writeverf;
12107         mutex_exit(&rp->r_statelock);

12109         error = nfs4_putpage(vp, poff, plen, B_ASYNC, cr, NULL);
12110         if (error == EAGAIN)
12111                 error = 0;

12113         /*
12114          * Do a flush based on the poff and plen arguments.  This
12115          * will synchronously write out any modified pages in the
12116          * range specified by (poff, plen) and wait until all of
12117          * the asynchronous i/o's in that range are done as well.
12118          */
12119         if (!error)
12120                 error = nfs4_putpage(vp, poff, plen, 0, cr, NULL);

12122         if (error)
12123                 return (error);

12125         mutex_enter(&rp->r_statelock);
12126         if (rp->r_writeverf != write_verf) {
12127                 mutex_exit(&rp->r_statelock);
12128                 goto top;
12129         }
12130         mutex_exit(&rp->r_statelock);

12132         /*
12133          * Now commit any pages which might need to be committed.
12134          * If the error, NFS_VERF_MISMATCH, is returned, then
12135          * start over with the flush operation.
12136          */
```

```
12137         error = nfs4_commit_vp(vp, poff, plen, cr, NFS4_WRITE_WAIT);

12139         if (error == NFS_VERF_MISMATCH)
12140                 goto top;

12142         return (error);
12143 }

12145 /*
12146  * nfs4_commit_vp()  will wait for other pending commits and
12147  * will either commit the whole file or a range, plen dictates
12148  * if we commit whole file. a value of zero indicates the whole
12149  * file. Called from nfs4_putpage_commit() or nfs4_sync_putapage()
12150  */
12151 static int
12152 nfs4_commit_vp(vnode_t *vp, u_offset_t poff, size_t plen,
12153     cred_t *cr, int wait_on_writes)
12154 {
12155         rnode4_t *rp;
12156         page_t *plist;
12157         offset3 offset;
12158         count3 len;

12160         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12162         rp = VTOR4(vp);

12164         /*
12165          *  before we gather commitable pages make
12166          *  sure there are no outstanding async writes
12167          */
12168         if (rp->r_count && wait_on_writes == NFS4_WRITE_WAIT) {
12169                 mutex_enter(&rp->r_statelock);
12170                 while (rp->r_count > 0) {
12171                         cv_wait(&rp->r_cv, &rp->r_statelock);
12172                 }
12173                 mutex_exit(&rp->r_statelock);
12174         }

12176         /*
12177          * Set the 'commit inprogress' state bit.  We must
12178          * first wait until any current one finishes.
12179          */
12180         mutex_enter(&rp->r_statelock);
12181         while (rp->r_flags & R4COMMIT) {
12182                 rp->r_flags |= R4COMMITWAIT;
12183                 cv_wait(&rp->r_commit.c_cv, &rp->r_statelock);
12184                 rp->r_flags &= ~R4COMMITWAIT;
12185         }
12186         rp->r_flags |= R4COMMIT;
12187         mutex_exit(&rp->r_statelock);

12189         /*
12190          * Gather all of the pages which need to be
12191          * committed.
12192          */
12193         if (plen == 0)
12194                 nfs4_get_commit(vp);
12195         else
12196                 nfs4_get_commit_range(vp, poff, plen);

12198         /*
12199          * Clear the 'commit inprogress' bit and disconnect the
12200          * page list which was gathered by nfs4_get_commit.
12201          */
12202         plist = rp->r_commit.c_pages;
```



```
12203         rp->r_commit.c_pages = NULL;
12204         offset = rp->r_commit.c_commbase;
12205         len = rp->r_commit.c_commlen;
12206         mutex_enter(&rp->r_statelock);
12207         rp->r_flags &= ~R4COMMIT;
12208         cv_broadcast(&rp->r_commit.c_cv);
12209         mutex_exit(&rp->r_statelock);

12211         /*
12212          * If any pages need to be committed, commit them and
12213          * then unlock them so that they can be freed some
12214          * time later.
12215          */
12216         if (plist == NULL)
12217                 return (0);

12219         /*
12220          * No error occurred during the flush portion
12221          * of this operation, so now attempt to commit
12222          * the data to stable storage on the server.
12223          *
12224          * This will unlock all of the pages on the list.
12225          */
12226         return (nfs4_sync_commit(vp, plist, offset, len, cr));
12227 }

12229 static int
12230 nfs4_sync_commit(vnode_t *vp, page_t *plist, offset3 offset, count3 count,
12231     cred_t *cr)
12232 {
12233         int error;
12234         page_t *pp;

12236         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12238         error = nfs4_commit(vp, (offset4)offset, (count3)count, cr);

12240         /*
12241          * If we got an error, then just unlock all of the pages
12242          * on the list.
12243          */
12244         if (error) {
12245                 while (plist != NULL) {
12246                         pp = plist;
12247                         page_sub(&plist, pp);
12248                         page_unlock(pp);
12249                 }
12250                 return (error);
12251         }
12252         /*
12253          * We've tried as hard as we can to commit the data to stable
12254          * storage on the server.  We just unlock the pages and clear
12255          * the commit required state.  They will get freed later.
12256          */
12257         while (plist != NULL) {
12258                 pp = plist;
12259                 page_sub(&plist, pp);
12260                 pp->p_fsdata = C_NOCOMMIT;
12261                 page_unlock(pp);
12262         }

12264         return (error);
12265 }

12267 static void
12268 do_nfs4_async_commit(vnode_t *vp, page_t *plist, offset3 offset, count3 count,
```

```
12269     cred_t *cr)
12270 {

12272         (void) nfs4_sync_commit(vp, plist, offset, count, cr);
12273 }

12275 /*ARGSUSED*/
12276 static int
12277 nfs4_setsecattr(vnode_t *vp, vsecattr_t *vsecattr, int flag, cred_t *cr,
12278         caller_context_t *ct)
12279 {
12280         int             error = 0;
12281         mntinfo4_t      *mi;
12282         vattr_t         va;
12283         vsecattr_t      nfsace4_vsap;

12285         mi = VTOMI4(vp);
12286         if (nfs_zone() != mi->mi_zone)
12287                 return (EIO);
12288         if (mi->mi_flags & MI4_ACL) {
12289                 /* if we have a delegation, return it */
12290                 if (VTOR4(vp)->r_deleg_type != OPEN_DELEGATE_NONE)
12291                         (void) nfs4delegreturn(VTOR4(vp),
12292                             NFS4_DR_REOPEN|NFS4_DR_PUSH);

12294                 error = nfs4_is_acl_mask_valid(vsecattr->vsa_mask,
12295                     NFS4_ACL_SET);
12296                 if (error) /* EINVAL */
12297                         return (error);

12299                 if (vsecattr->vsa_mask & (VSA_ACL | VSA_DFACL)) {
12300                         /*
12301                          * These are aclent_t type entries.
12302                          */
12303                         error = vs_aent_to_ace4(vsecattr, &nfsace4_vsap,
12304                             vp->v_type == VDIR, FALSE);
12305                         if (error)
12306                                 return (error);
12307                 } else {
12308                         /*
12309                          * These are ace_t type entries.
12310                          */
12311                         error = vs_acet_to_ace4(vsecattr, &nfsace4_vsap,
12312                             FALSE);
12313                         if (error)
12314                                 return (error);
12315                 }
12316                 bzero(&va, sizeof (va));
12317                 error = nfs4setattr(vp, &va, flag, cr, &nfsace4_vsap);
12318                 vs_ace4_destroy(&nfsace4_vsap);
12319                 return (error);
12320         }
12321         return (ENOSYS);
12322 }

12324 /* ARGSUSED */
12325 int
12326 nfs4_getsecattr(vnode_t *vp, vsecattr_t *vsecattr, int flag, cred_t *cr,
12327         caller_context_t *ct)
12328 {
12329         int             error;
12330         mntinfo4_t      *mi;
12331         nfs4_ga_res_t   gar;
12332         rnode4_t        *rp = VTOR4(vp);

12334         mi = VTOMI4(vp);
```



```
12335         if (nfs_zone() != mi->mi_zone)
12336                 return (EIO);

12338         bzero(&gar, sizeof (gar));
12339         gar.n4g_vsa.vsa_mask = vsecattr->vsa_mask;

12341         /*
12342          * vsecattr->vsa_mask holds the original acl request mask.
12343          * This is needed when determining what to return.
12344          * (See: nfs4_create_getsecattr_return())
12345          */
12346         error = nfs4_is_acl_mask_valid(vsecattr->vsa_mask, NFS4_ACL_GET);
12347         if (error) /* EINVAL */
12348                 return (error);

12350         /*
12351          * If this is a referral stub, don't try to go OTW for an ACL
12352          */
12353         if (RP_ISSTUB_REFERRAL(VTOR4(vp)))
12354                 return (fs_fab_acl(vp, vsecattr, flag, cr, ct));

12356         if (mi->mi_flags & MI4_ACL) {
12357                 /*
12358                  * Check if the data is cached and the cache is valid.  If it
12359                  * is we don't go over the wire.
12360                  */
12361                 if (rp->r_secattr != NULL && ATTRCACHE4_VALID(vp)) {
12362                         mutex_enter(&rp->r_statelock);
12363                         if (rp->r_secattr != NULL) {
12364                                 error = nfs4_create_getsecattr_return(
12365                                     rp->r_secattr, vsecattr, rp->r_attr.va_uid,
12366                                     rp->r_attr.va_gid,
12367                                     vp->v_type == VDIR);
12368                                 if (!error) { /* error == 0 - Success! */
12369                                         mutex_exit(&rp->r_statelock);
12370                                         return (error);
12371                                 }
12372                         }
12373                         mutex_exit(&rp->r_statelock);
12374                 }

12376                 /*
12377                  * The getattr otw call will always get both the acl, in
12378                  * the form of a list of nfsace4's, and the number of acl
12379                  * entries; independent of the value of gar.n4g_vsa.vsa_mask.
12380                  */
12381                 gar.n4g_va.va_mask = AT_ALL;
12382                 error =  nfs4_getattr_otw(vp, &gar, cr, 1);
12383                 if (error) {
12384                         vs_ace4_destroy(&gar.n4g_vsa);
12385                         if (error == ENOTSUP || error == EOPNOTSUPP)
12386                                 error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12387                         return (error);
12388                 }

12390                 if (!(gar.n4g_resbmap & FATTR4_ACL_MASK)) {
12391                         /*
12392                          * No error was returned, but according to the response
12393                          * bitmap, neither was an acl.
12394                          */
12395                         vs_ace4_destroy(&gar.n4g_vsa);
12396                         error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12397                         return (error);
12398                 }

12400                 /*
```

```
12401                  * Update the cache with the ACL.
12402                  */
12403                 nfs4_acl_fill_cache(rp, &gar.n4g_vsa);

12405                 error = nfs4_create_getsecattr_return(&gar.n4g_vsa,
12406                     vsecattr, gar.n4g_va.va_uid, gar.n4g_va.va_gid,
12407                     vp->v_type == VDIR);
12408                 vs_ace4_destroy(&gar.n4g_vsa);
12409                 if ((error) && (vsecattr->vsa_mask &
12410                     (VSA_ACL | VSA_ACLCNT | VSA_DFACL | VSA_DFACLCNT)) &&
12411                     (error != EACCES)) {
12412                         error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12413                 }
12414                 return (error);
12415         }
12416         error = fs_fab_acl(vp, vsecattr, flag, cr, ct);
12417         return (error);
12418 }

12420 /*
12421  * The function returns:
12422  *      - 0 (zero) if the passed in "acl_mask" is a valid request.
12423  *      - EINVAL if the passed in "acl_mask" is an invalid request.
12424  *
12425  * In the case of getting an acl (op == NFS4_ACL_GET) the mask is invalid if:
12426  * - We have a mixture of ACE and ACL requests (e.g. VSA_ACL | VSA_ACE)
12427  *
12428  * In the case of setting an acl (op == NFS4_ACL_SET) the mask is invalid if:
12429  * - We have a mixture of ACE and ACL requests (e.g. VSA_ACL | VSA_ACE)
12430  * - We have a count field set without the corresponding acl field set. (e.g. -
12431  * VSA_ACECNT is set, but VSA_ACE is not)
12432  */
12433 static int
12434 nfs4_is_acl_mask_valid(uint_t acl_mask, nfs4_acl_op_t op)
12435 {
12436         /* Shortcut the masks that are always valid. */
12437         if (acl_mask == (VSA_ACE | VSA_ACECNT))
12438                 return (0);
12439         if (acl_mask == (VSA_ACL | VSA_ACLCNT | VSA_DFACL | VSA_DFACLCNT))
12440                 return (0);

12442         if (acl_mask & (VSA_ACE | VSA_ACECNT)) {
12443                 /*
12444                  * We can't have any VSA_ACL type stuff in the mask now.
12445                  */
12446                 if (acl_mask & (VSA_ACL | VSA_ACLCNT | VSA_DFACL |
12447                     VSA_DFACLCNT))
12448                         return (EINVAL);

12450                 if (op == NFS4_ACL_SET) {
12451                         if ((acl_mask & VSA_ACECNT) && !(acl_mask & VSA_ACE))
12452                                 return (EINVAL);
12453                 }
12454         }

12456         if (acl_mask & (VSA_ACL | VSA_ACLCNT | VSA_DFACL | VSA_DFACLCNT)) {
12457                 /*
12458                  * We can't have any VSA_ACE type stuff in the mask now.
12459                  */
12460                 if (acl_mask & (VSA_ACE | VSA_ACECNT))
12461                         return (EINVAL);

12463                 if (op == NFS4_ACL_SET) {
12464                         if ((acl_mask & VSA_ACLCNT) && !(acl_mask & VSA_ACL))
12465                                 return (EINVAL);
```



```
12467                         if ((acl_mask & VSA_DFACLCNT) &&
12468                             !(acl_mask & VSA_DFACL))
12469                                 return (EINVAL);
12470                 }
12471         }
12472         return (0);
12473 }

12475 /*
12476  * The theory behind creating the correct getsecattr return is simply this:
12477  * "Don't return anything that the caller is not expecting to have to free."
12478  */
12479 static int
12480 nfs4_create_getsecattr_return(vsecattr_t *filled_vsap, vsecattr_t *vsap,
12481     uid_t uid, gid_t gid, int isdir)
12482 {
12483         int error = 0;
12484         /* Save the mask since the translators modify it. */
12485         uint_t  orig_mask = vsap->vsa_mask;

12487         if (orig_mask & (VSA_ACE | VSA_ACECNT)) {
12488                 error = vs_ace4_to_acet(filled_vsap, vsap, uid, gid, FALSE);

12490                 if (error)
12491                         return (error);

12493                 /*
12494                  * If the caller only asked for the ace count (VSA_ACECNT)
12495                  * don't give them the full acl (VSA_ACE), free it.
12496                  */
12497                 if (!orig_mask & VSA_ACE) {
12498                         if (vsap->vsa_aclentp != NULL) {
12499                                 kmem_free(vsap->vsa_aclentp,
12500                                     vsap->vsa_aclcnt * sizeof (ace_t));
12501                                 vsap->vsa_aclentp = NULL;
12502                         }
12503                 }
12504                 vsap->vsa_mask = orig_mask;

12506         } else if (orig_mask & (VSA_ACL | VSA_ACLCNT | VSA_DFACL |
12507             VSA_DFACLCNT)) {
12508                 error = vs_ace4_to_aent(filled_vsap, vsap, uid, gid,
12509                     isdir, FALSE);

12511                 if (error)
12512                         return (error);

12514                 /*
12515                  * If the caller only asked for the acl count (VSA_ACLCNT)
12516                  * and/or the default acl count (VSA_DFACLCNT) don't give them
12517                  * the acl (VSA_ACL) or default acl (VSA_DFACL), free it.
12518                  */
12519                 if (!orig_mask & VSA_ACL) {
12520                         if (vsap->vsa_aclentp != NULL) {
12521                                 kmem_free(vsap->vsa_aclentp,
12522                                     vsap->vsa_aclcnt * sizeof (aclent_t));
12523                                 vsap->vsa_aclentp = NULL;
12524                         }
12525                 }

12527                 if (!orig_mask & VSA_DFACL) {
12528                         if (vsap->vsa_dfaclentp != NULL) {
12529                                 kmem_free(vsap->vsa_dfaclentp,
12530                                     vsap->vsa_dfaclcnt * sizeof (aclent_t));
12531                                 vsap->vsa_dfaclentp = NULL;
12532                         }
```

```
12533                 }
12534                 vsap->vsa_mask = orig_mask;
12535         }
12536         return (0);
12537 }

12539 /* ARGSUSED */
12540 int
12541 nfs4_shrlock(vnode_t *vp, int cmd, struct shrlock *shr, int flag, cred_t *cr,
12542     caller_context_t *ct)
12543 {
12544         int error;

12546         if (nfs_zone() != VTOMI4(vp)->mi_zone)
12547                 return (EIO);
12548         /*
12549          * check for valid cmd parameter
12550          */
12551         if (cmd != F_SHARE && cmd != F_UNSHARE && cmd != F_HASREMOTELOCKS)
12552                 return (EINVAL);

12554         /*
12555          * Check access permissions
12556          */
12557         if ((cmd & F_SHARE) &&
12558             (((shr->s_access & F_RDACC) && (flag & FREAD) == 0) ||
12559             (shr->s_access == F_WRACC && (flag & FWRITE) == 0)))
12560                 return (EBADF);

12562         /*
12563          * If the filesystem is mounted using local locking, pass the
12564          * request off to the local share code.
12565          */
12566         if (VTOMI4(vp)->mi_flags & MI4_LLOCK)
12567                 return (fs_shrlock(vp, cmd, shr, flag, cr, ct));

12569         switch (cmd) {
12570         case F_SHARE:
12571         case F_UNSHARE:
12572                 /*
12573                  * This will be properly implemented later,
12574                  * see RFE: 4823948 .
12575                  */
12576                 error = EAGAIN;
12577                 break;

12579         case F_HASREMOTELOCKS:
12580                 /*
12581                  * NFS client can't store remote locks itself
12582                  */
12583                 shr->s_access = 0;
12584                 error = 0;
12585                 break;

12587         default:
12588                 error = EINVAL;
12589                 break;
12590         }

12592         return (error);
12593 }

12595 /*
12596  * Common code called by directory ops to update the attrcache
12597  */
12598 static int
```



```
12599 nfs4_update_attrcache(nfsstat4 status, nfs4_ga_res_t *garp,
12600     hrtime_t t, vnode_t *vp, cred_t *cr)
12601 {
12602         int error = 0;

12604         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12606         if (status != NFS4_OK) {
12607                 /* getattr not done or failed */
12608                 PURGE_ATTRCACHE4(vp);
12609                 return (error);
12610         }

12612         if (garp) {
12613                 nfs4_attr_cache(vp, garp, t, cr, FALSE, NULL);
12614         } else {
12615                 PURGE_ATTRCACHE4(vp);
12616         }
12617         return (error);
12618 }

12620 /*
12621  * Update directory caches for directory modification ops (link, rename, etc.)
12622  * When dinfo is NULL, manage dircaches in the old way.
12623  */
12624 static void
12625 nfs4_update_dircaches(change_info4 *cinfo, vnode_t *dvp, vnode_t *vp, char *nm,
12626     dirattr_info_t *dinfo)
12627 {
12628         rnode4_t        *drp = VTOR4(dvp);

12630         ASSERT(nfs_zone() == VTOMI4(dvp)->mi_zone);

12632         /* Purge rddir cache for dir since it changed */
12633         if (drp->r_dir != NULL)
12634                 nfs4_purge_rddir_cache(dvp);

12636         /*
12637          * If caller provided dinfo, then use it to manage dir caches.
12638          */
12639         if (dinfo != NULL) {
12640                 if (vp != NULL) {
12641                         mutex_enter(&VTOR4(vp)->r_statev4_lock);
12642                         if (!VTOR4(vp)->created_v4) {
12643                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12644                                 dnlc_update(dvp, nm, vp);
12645                         } else {
12646                                 /*
12647                                  * XXX don't update if the created_v4 flag is
12648                                  * set
12649                                  */
12650                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12651                                 NFS4_DEBUG(nfs4_client_state_debug,
12652                                     (CE_NOTE, "nfs4_update_dircaches: "
12653                                     "don't update dnlc: created_v4 flag"));
12654                         }
12655                 }

12657                 nfs4_attr_cache(dvp, dinfo->di_garp, dinfo->di_time_call,
12658                     dinfo->di_cred, FALSE, cinfo);

12660                 return;
12661         }

12663         /*
12664          * Caller didn't provide dinfo, then check change_info4 to update DNLC.
```

```
12665          * Since caller modified dir but didn't receive post-dirmod-op dir
12666          * attrs, the dir's attrs must be purged.
12667          *
12668          * XXX this check and dnlc update/purge should really be atomic,
12669          * XXX but can't use rnode statelock because it'll deadlock in
12670          * XXX dnlc_purge_vp, however, the risk is minimal even if a race
12671          * XXX does occur.
12672          *
12673          * XXX We also may want to check that atomic is true in the
12674          * XXX change_info struct. If it is not, the change_info may
12675          * XXX reflect changes by more than one clients which means that
12676          * XXX our cache may not be valid.
12677          */
12678         PURGE_ATTRCACHE4(dvp);
12679         if (drp->r_change == cinfo->before) {
12680                 /* no changes took place in the directory prior to our link */
12681                 if (vp != NULL) {
12682                         mutex_enter(&VTOR4(vp)->r_statev4_lock);
12683                         if (!VTOR4(vp)->created_v4) {
12684                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12685                                 dnlc_update(dvp, nm, vp);
12686                         } else {
12687                                 /*
12688                                  * XXX dont' update if the created_v4 flag
12689                                  * is set
12690                                  */
12691                                 mutex_exit(&VTOR4(vp)->r_statev4_lock);
12692                                 NFS4_DEBUG(nfs4_client_state_debug, (CE_NOTE,
12693                                     "nfs4_update_dircaches: don't"
12694                                     " update dnlc: created_v4 flag"));
12695                         }
12696                 }
12697         } else {
12698                 /* Another client modified directory - purge its dnlc cache */
12699                 dnlc_purge_vp(dvp);
12700         }
12701 }

12703 /*
12704  * The OPEN_CONFIRM operation confirms the sequence number used in OPENing a
12705  * file.
12706  *
12707  * The 'reopening_file' boolean should be set to TRUE if we are reopening this
12708  * file (ie: client recovery) and otherwise set to FALSE.
12709  *
12710  * 'nfs4_start/end_op' should have been called by the proper (ie: not recovery
12711  * initiated) calling functions.
12712  *
12713  * 'resend' is set to TRUE if this is a OPEN_CONFIRM issued as a result
12714  * of resending a 'lost' open request.
12715  *
12716  * 'num_bseqid_retryp' makes sure we don't loop forever on a broken
12717  * server that hands out BAD_SEQID on open confirm.
12718  *
12719  * Errors are returned via the nfs4_error_t parameter.
12720  */
12721 void
12722 nfs4open_confirm(vnode_t *vp, seqid4 *seqid, stateid4 *stateid, cred_t *cr,
12723     bool_t reopening_file, bool_t *retry_open, nfs4_open_owner_t *oop,
12724     bool_t resend, nfs4_error_t *ep, int *num_bseqid_retryp)
12725 {
12726         COMPOUND4args_clnt args;
12727         COMPOUND4res_clnt res;
12728         nfs_argop4 argop[2];
12729         nfs_resop4 *resop;
12730         int doqueue = 1;
```



```
12731         mntinfo4_t *mi;
12732         OPEN_CONFIRM4args *open_confirm_args;
12733         int needrecov;

12735         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);
12736 #if DEBUG
12737         mutex_enter(&oop->oo_lock);
12738         ASSERT(oop->oo_seqid_inuse);
12739         mutex_exit(&oop->oo_lock);
12740 #endif

12742 recov_retry_confirm:
12743         nfs4_error_zinit(ep);
12744         *retry_open = FALSE;

12746         if (resend)
12747                 args.ctag = TAG_OPEN_CONFIRM_LOST;
12748         else
12749                 args.ctag = TAG_OPEN_CONFIRM;

12751         args.array_len = 2;
12752         args.array = argop;

12754         /* putfh target fh */
12755         argop[0].argop = OP_CPUTFH;
12756         argop[0].nfs_argop4_u.opcputfh.sfh = VTOR4(vp)->r_fh;

12758         argop[1].argop = OP_OPEN_CONFIRM;
12759         open_confirm_args = &argop[1].nfs_argop4_u.opopen_confirm;

12761         (*seqid) += 1;
12762         open_confirm_args->seqid = *seqid;
12763         open_confirm_args->open_stateid = *stateid;

12765         mi = VTOMI4(vp);

12767         rfs4call(mi, &args, &res, cr, &doqueue, 0, ep);

12769         if (!ep->error && nfs4_need_to_bump_seqid(&res)) {
12770                 nfs4_set_open_seqid((*seqid), oop, args.ctag);
12771         }

12773         needrecov = nfs4_needs_recovery(ep, FALSE, mi->mi_vfsp);
12774         if (!needrecov && ep->error)
12775                 return;

12777         if (needrecov) {
12778                 bool_t abort = FALSE;

12780                 if (reopening_file == FALSE) {
12781                         nfs4_bseqid_entry_t *bsep = NULL;

12783                         if (!ep->error && res.status == NFS4ERR_BAD_SEQID)
12784                                 bsep = nfs4_create_bseqid_entry(oop, NULL,
12785                                     vp, 0, args.ctag,
12786                                     open_confirm_args->seqid);

12788                         abort = nfs4_start_recovery(ep, VTOMI4(vp), vp, NULL,
12789                             NULL, NULL, OP_OPEN_CONFIRM, bsep, NULL, NULL);
12790                         if (bsep) {
12791                                 kmem_free(bsep, sizeof (*bsep));
12792                                 if (num_bseqid_retryp &&
12793                                     --(*num_bseqid_retryp) == 0)
12794                                         abort = TRUE;
12795                         }
12796                 }
```

```
12797                 if ((ep->error == ETIMEDOUT ||
12798                     res.status == NFS4ERR_RESOURCE) &&
12799                     abort == FALSE && resend == FALSE) {
12800                         if (!ep->error)
12801                                 (void) xdr_free(xdr_COMPOUND4res_clnt,
12802                                     (caddr_t)&res);

12804                         delay(SEC_TO_TICK(confirm_retry_sec));
12805                         goto recov_retry_confirm;
12806                 }
12807                 /* State may have changed so retry the entire OPEN op */
12808                 if (abort == FALSE)
12809                         *retry_open = TRUE;
12810                 else
12811                         *retry_open = FALSE;
12812                 if (!ep->error)
12813                         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
12814                 return;
12815         }

12817         if (res.status) {
12818                 (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
12819                 return;
12820         }

12822         resop = &res.array[1];  /* open confirm res */
12823         bcopy(&resop->nfs_resop4_u.opopen_confirm.open_stateid,
12824             stateid, sizeof (*stateid));

12826         (void) xdr_free(xdr_COMPOUND4res_clnt, (caddr_t)&res);
12827 }

12829 /*
12830  * Return the credentials associated with a client state object.  The
12831  * caller is responsible for freeing the credentials.
12832  */

12834 static cred_t *
12835 state_to_cred(nfs4_open_stream_t *osp)
12836 {
12837         cred_t *cr;

12839         /*
12840          * It's ok to not lock the open stream and open owner to get
12841          * the oo_cred since this is only written once (upon creation)
12842          * and will not change.
12843          */
12844         cr = osp->os_open_owner->oo_cred;
12845         crhold(cr);

12847         return (cr);
12848 }

12850 /*
12851  * nfs4_find_sysid
12852  *
12853  * Find the sysid for the knetconfig associated with the given mi.
12854  */
12855 static struct lm_sysid *
12856 nfs4_find_sysid(mntinfo4_t *mi)
12857 {
12858         ASSERT(nfs_zone() == mi->mi_zone);

12860         /*
12861          * Switch from RDMA knconf to original mount knconf
12862          */
```



```
12863         return (lm_get_sysid(ORIG_KNCONF(mi), &mi->mi_curr_serv->sv_addr,
12864             mi->mi_curr_serv->sv_hostname, NULL));
12865 }

12867 #ifdef DEBUG
12868 /*
12869  * Return a string version of the call type for easy reading.
12870  */
12871 static char *
12872 nfs4frlock_get_call_type(nfs4_lock_call_type_t ctype)
12873 {
12874         switch (ctype) {
12875         case NFS4_LCK_CTYPE_NORM:
12876                 return ("NORMAL");
12877         case NFS4_LCK_CTYPE_RECLAIM:
12878                 return ("RECLAIM");
12879         case NFS4_LCK_CTYPE_RESEND:
12880                 return ("RESEND");
12881         case NFS4_LCK_CTYPE_REINSTATE:
12882                 return ("REINSTATE");
12883         default:
12884                 cmn_err(CE_PANIC, "nfs4frlock_get_call_type: got illegal "
12885                     "type %d", ctype);
12886                 return ("");
12887         }
12888 }
12889 #endif

12891 /*
12892  * Map the frlock cmd and lock type to the NFSv4 over-the-wire lock type
12893  * Unlock requests don't have an over-the-wire locktype, so we just return
12894  * something non-threatening.
12895  */

12897 static nfs_lock_type4
12898 flk_to_locktype(int cmd, int l_type)
12899 {
12900         ASSERT(l_type == F_RDLCK || l_type == F_WRLCK || l_type == F_UNLCK);

12902         switch (l_type) {
12903         case F_UNLCK:
12904                 return (READ_LT);
12905         case F_RDLCK:
12906                 if (cmd == F_SETLK)
12907                         return (READ_LT);
12908                 else
12909                         return (READW_LT);
12910         case F_WRLCK:
12911                 if (cmd == F_SETLK)
12912                         return (WRITE_LT);
12913                 else
12914                         return (WRITEW_LT);
12915         }
12916         panic("flk_to_locktype");
12917         /*NOTREACHED*/
12918 }

12920 /*
12921  * Do some preliminary checks for nfs4frlock.
12922  */
12923 static int
12924 nfs4frlock_validate_args(int cmd, flock64_t *flk, int flag, vnode_t *vp,
12925     u_offset_t offset)
12926 {
12927         int error = 0;
```

```
12929         /*
12930          * If we are setting a lock, check that the file is opened
12931          * with the correct mode.
12932          */
12933         if (cmd == F_SETLK || cmd == F_SETLKW) {
12934                 if ((flk->l_type == F_RDLCK && (flag & FREAD) == 0) ||
12935                     (flk->l_type == F_WRLCK && (flag & FWRITE) == 0)) {
12936                         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
12937                             "nfs4frlock_validate_args: file was opened with "
12938                             "incorrect mode"));
12939                         return (EBADF);
12940                 }
12941         }

12943         /* Convert the offset. It may need to be restored before returning. */
12944         if (error = convoff(vp, flk, 0, offset)) {
12945                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
12946                     "nfs4frlock_validate_args: convoff  =>  error= %d\n",
12947                     error));
12948                 return (error);
12949         }

12951         return (error);
12952 }

12954 /*
12955  * Set the flock64's lm_sysid for nfs4frlock.
12956  */
12957 static int
12958 nfs4frlock_get_sysid(struct lm_sysid **lspp, vnode_t *vp, flock64_t *flk)
12959 {
12960         ASSERT(nfs_zone() == VTOMI4(vp)->mi_zone);

12962         /* Find the lm_sysid */
12963         *lspp = nfs4_find_sysid(VTOMI4(vp));

12965         if (*lspp == NULL) {
12966                 NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE,
12967                     "nfs4frlock_get_sysid: no sysid, return ENOLCK"));
12968                 return (ENOLCK);
12969         }

12971         flk->l_sysid = lm_sysidt(*lspp);

12973         return (0);
12974 }

12976 /*
12977  * Do the remaining preliminary setup for nfs4frlock.
12978  */
12979 static void
12980 nfs4frlock_pre_setup(clock_t *tick_delayp, nfs4_recov_state_t *recov_statep,
12981     flock64_t *flk, short *whencep, vnode_t *vp, cred_t *search_cr,
12982     cred_t **cred_otw)
12983 {
12984         /*
12985          * set tick_delay to the base delay time.
12986          * (nfs4_base_wait_time is in msecs)
 110          * (NFS4_BASE_WAIT_TIME is in secs)
12987          */

12989         *tick_delayp = drv_usectohz(nfs4_base_wait_time * 1000);
 113         *tick_delayp = drv_usectohz(NFS4_BASE_WAIT_TIME * 1000 * 1000);

12991         /*
12992          * If lock is relative to EOF, we need the newest length of the
```



```
12993          * file. Therefore invalidate the ATTR_CACHE.
12994          */

12996         *whencep = flk->l_whence;

12998         if (*whencep == 2)              /* SEEK_END */
12999                 PURGE_ATTRCACHE4(vp);

13001         recov_statep->rs_flags = 0;
13002         recov_statep->rs_num_retry_despite_err = 0;
13003         *cred_otw = nfs4_get_otw_cred(search_cr, VTOMI4(vp), NULL);
13004 }
______unchanged_portion_omitted_

14757 /*
14758  * Wait for 'tick_delay' clock ticks.
14759  * Implement exponential backoff until hit the lease_time of this nfs4_server.
14760  *
14761  * The client should retry to acquire the lock faster than the lease period.
14762  * We use roughly half of the lease time to use a similar calculation as it is
14763  * used in nfs4_renew_lease_thread().
1884  * NOTE: lock_lease_time is in seconds.
14764  *
14765  * XXX For future improvements, should implement a waiting queue scheme.
14766  */
14767 static int
14768 nfs4_block_and_wait(clock_t *tick_delay, rnode4_t *rp)
14769 {
14770         long max_msec_delay = 1 * 1000;         /* 1 sec */
14771         nfs4_server_t *sp;
14772         mntinfo4_t *mi = VTOMI4(RTOV4(rp));
1891         long milliseconds_delay;
1892         time_t lock_lease_time;

14774         /* wait tick_delay clock ticks or siginteruptus */
14775         if (delay_sig(*tick_delay)) {
14776                 return (EINTR);
14777         }

14779 #endif /* ! codereview */
14780         NFS4_DEBUG(nfs4_client_lock_debug, (CE_NOTE, "nfs4_block_and_wait: "
14781             "reissue the lock request: blocked for %ld clock ticks: %ld "
14782             "milliseconds", *tick_delay, drv_hztousec(*tick_delay) / 1000));

14784         /*
14785          * Get the current lease time and propagation time for the server
14786          * associated with the given file. Note that both times could
14787          * change immediately after this section.
14788          */
14789         nfs_rw_enter_sig(&mi->mi_recovlock, RW_READER, 0);
14790         sp = find_nfs4_server(mi);
14791         if (sp != NULL) {
14792                 if (!(mi->mi_vfsp->vfs_flag & VFS_UNMOUNTED)) {
14793                         max_msec_delay = sp->s_lease_time * 1000 / 2 -
14794                                          (3 * sp->propagation_delay.tv_sec *
14795                                          1000);
14796                 }
14797                 mutex_exit(&sp->s_lock);
14798                 nfs4_server_rele(sp);
14799         }
14800         nfs_rw_exit(&mi->mi_recovlock);
1898         /* get the lease time */
1899         lock_lease_time = r2lease_time(rp);

14802         max_msec_delay = MAX(max_msec_delay, nfs4_base_wait_time);
14803         *tick_delay = MIN(drv_usectohz(max_msec_delay * 1000), *tick_delay * 2);
```

```
1901         /* drv_hztousec converts ticks to microseconds */
1902         milliseconds_delay = drv_hztousec(*tick_delay) / 1000;
1903         if (milliseconds_delay < lock_lease_time * 1000) {
1904                 *tick_delay = 2 * *tick_delay;
1905                 if (drv_hztousec(*tick_delay) > lock_lease_time * 1000 * 1000)
1906                         *tick_delay = drv_usectohz(lock_lease_time*1000*1000);
1907         }
14804         return (0);
14805 }

14807 void
14808 nfs4_vnops_init(void)
14809 {
14810 }
```

_____unchanged_portion_omitted_

```
**********************************************************
   19631 Mon May 12 10:06:22 2014
new/usr/src/uts/common/nfs/rnode4.h
4827 nfs4: slow file locking
4837 NFSv4 client lock retry delay upper limit should be shorter
**********************************************************
______unchanged_portion_omitted_

 388 #ifdef _KERNEL

 390 extern long nrnode;

 392 /* Used for r_delay_interval */
 393 #define NFS4_INITIAL_DELAY_INTERVAL      1
 394 #define NFS4_MAX_DELAY_INTERVAL         20

 396 /* Used for check_rtable4 */
 397 #define NFSV4_RTABLE4_OK                 0
 398 #define NFSV4_RTABLE4_NOT_FREE_LIST      1
 399 #define NFSV4_RTABLE4_DIRTY_PAGES        2
 400 #define NFSV4_RTABLE4_POS_R_COUNT        3

 402 extern rnode4_t *r4find(r4hashq_t *, nfs4_sharedfh_t *, struct vfs *);
 403 extern rnode4_t *r4find_unlocked(nfs4_sharedfh_t *, struct vfs *);
 404 extern void     r4flush(struct vfs *, cred_t *);
 405 extern void     destroy_rtable4(struct vfs *, cred_t *);
 406 extern int      check_rtable4(struct vfs *);
 407 extern void     rp4_addfree(rnode4_t *, cred_t *);
 408 extern void     rp4_addhash(rnode4_t *);
 409 extern void     rp4_rmhash(rnode4_t *);
 410 extern void     rp4_rmhash_locked(rnode4_t *);
 411 extern int      rtable4hash(nfs4_sharedfh_t *);

 413 extern vnode_t *makenfs4node(nfs4_sharedfh_t *, nfs4_ga_res_t *, struct vfs *,
 414                              hrtime_t, cred_t *, vnode_t *, nfs4_fname_t *);
 415 extern vnode_t *makenfs4node_by_fh(nfs4_sharedfh_t *, nfs4_sharedfh_t *,
 416     nfs4_fname_t **, nfs4_ga_res_t *, mntinfo4_t *, cred_t *, hrtime_t);

 418 extern nfs4_opinst_t *r4mkopenlist(struct mntinfo4 *);
 419 extern void     r4releopenlist(nfs4_opinst_t *);
 420 extern int      r4find_by_fsid(mntinfo4_t *, fattr4_fsid *);

 422 /* Access cache calls */
 423 extern nfs4_access_type_t nfs4_access_check(rnode4_t *, uint32_t, cred_t *);
 424 extern void     nfs4_access_cache(rnode4_t *rp, uint32_t, uint32_t, cred_t *);
 425 extern int      nfs4_access_purge_rp(rnode4_t *);

 427 extern int      nfs4_free_data_reclaim(rnode4_t *);
 428 extern void     nfs4_rnode_invalidate(struct vfs *);

 430 extern time_t   r2lease_time(rnode4_t *);
 430 extern int      nfs4_directio(vnode_t *, int, cred_t *);

 432 /* shadow vnode functions */
 433 extern void     sv_activate(vnode_t **, vnode_t *, nfs4_fname_t **, int);
 434 extern vnode_t  *sv_find(vnode_t *, vnode_t *, nfs4_fname_t **);
 435 extern void     sv_update_path(vnode_t *, char *, char *);
 436 extern void     sv_inactive(vnode_t *);
 437 extern void     sv_exchange(vnode_t **);
 438 extern void     sv_uninit(svnode_t *);
 439 extern void     nfs4_clear_open_streams(rnode4_t *);

 441 /*
 442  * Mark cached attributes as timed out
 443  *
 444  * The caller must not be holding the rnode r_statelock mutex.
```



```
 445  */
 446 #define PURGE_ATTRCACHE4_LOCKED(rp)                             \
 447         rp->r_time_attr_inval = gethrtime();                    \
 448         rp->r_time_attr_saved = rp->r_time_attr_inval;          \
 449         rp->r_pathconf.pc4_xattr_valid = 0;                     \
 450         rp->r_pathconf.pc4_cache_valid = 0;

 452 #define PURGE_ATTRCACHE4(vp)    {                               \
 453         rnode4_t *rp = VTOR4(vp);                               \
 454         mutex_enter(&rp->r_statelock);                          \
 455         PURGE_ATTRCACHE4_LOCKED(rp);                            \
 456         mutex_exit(&rp->r_statelock);                           \
 457 }
______unchanged_portion_omitted_
```