```
**********************************************************
  134159 Wed May 25 09:58:21 2016
new/usr/src/uts/common/fs/zfs/zfs_vnops.c
6940 Cannot unlink directories when over quota
**********************************************************
______unchanged_portion_omitted_

2029 /*
2030  * Remove a directory subdir entry.  If the current working
2031  * directory is the same as the subdir to be removed, the
2032  * remove will fail.
2033  *
2034  *      IN:     dvp     - vnode of directory to remove from.
2035  *              name    - name of directory to be removed.
2036  *              cwd     - vnode of current working directory.
2037  *              cr      - credentials of caller.
2038  *              ct      - caller context
2039  *              flags   - case flags
2040  *
2041  *      RETURN: 0 on success, error code on failure.
2042  *
2043  * Timestamps:
2044  *      dvp - ctime|mtime updated
2045  */
2046 /*ARGSUSED*/
2047 static int
2048 zfs_rmdir(vnode_t *dvp, char *name, vnode_t *cwd, cred_t *cr,
2049     caller_context_t *ct, int flags)
2050 {
2051         znode_t         *dzp = VTOZ(dvp);
2052         znode_t         *zp;
2053         vnode_t         *vp;
2054         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
2055         zilog_t         *zilog;
2056         zfs_dirlock_t   *dl;
2057         dmu_tx_t        *tx;
2058         int             error;
2059         int             zflg = ZEXISTS;
2060         boolean_t       waited = B_FALSE;

2062         ZFS_ENTER(zfsvfs);
2063         ZFS_VERIFY_ZP(dzp);
2064         zilog = zfsvfs->z_log;

2066         if (flags & FIGNORECASE)
2067                 zflg |= ZCILOOK;
2068 top:
2069         zp = NULL;

2071         /*
2072          * Attempt to lock directory; fail if entry doesn't exist.
2073          */
2074         if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
2075             NULL, NULL)) {
2076                 ZFS_EXIT(zfsvfs);
2077                 return (error);
2078         }

2080         vp = ZTOV(zp);

2082         if (error = zfs_zaccess_delete(dzp, zp, cr)) {
2083                 goto out;
2084         }

2086         if (vp->v_type != VDIR) {
2087                 error = SET_ERROR(ENOTDIR);
```



```
2088                 goto out;
2089         }

2091         if (vp == cwd) {
2092                 error = SET_ERROR(EINVAL);
2093                 goto out;
2094         }

2096         vnevent_rmdir(vp, dvp, name, ct);

2098         /*
2099          * Grab a lock on the directory to make sure that noone is
2100          * trying to add (or lookup) entries while we are removing it.
2101          */
2102         rw_enter(&zp->z_name_lock, RW_WRITER);

2104         /*
2105          * Grab a lock on the parent pointer to make sure we play well
2106          * with the treewalk and directory rename code.
2107          */
2108         rw_enter(&zp->z_parent_lock, RW_WRITER);

2110         tx = dmu_tx_create(zfsvfs->z_os);
2111         dmu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
2112         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
2113         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
2114         zfs_sa_upgrade_txholds(tx, zp);
2115         zfs_sa_upgrade_txholds(tx, dzp);
2116         dmu_tx_mark_netfree(tx);
2117 #endif /* ! codereview */
2118         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
2119         if (error) {
2120                 rw_exit(&zp->z_parent_lock);
2121                 rw_exit(&zp->z_name_lock);
2122                 zfs_dirent_unlock(dl);
2123                 VN_RELE(vp);
2124                 if (error == ERESTART) {
2125                         waited = B_TRUE;
2126                         dmu_tx_wait(tx);
2127                         dmu_tx_abort(tx);
2128                         goto top;
2129                 }
2130                 dmu_tx_abort(tx);
2131                 ZFS_EXIT(zfsvfs);
2132                 return (error);
2133         }

2135         error = zfs_link_destroy(dl, zp, tx, zflg, NULL);

2137         if (error == 0) {
2138                 uint64_t txtype = TX_RMDIR;
2139                 if (flags & FIGNORECASE)
2140                         txtype |= TX_CI;
2141                 zfs_log_remove(zilog, tx, txtype, dzp, name, ZFS_NO_OBJECT);
2142         }

2144         dmu_tx_commit(tx);

2146         rw_exit(&zp->z_parent_lock);
2147         rw_exit(&zp->z_name_lock);
2148 out:
2149         zfs_dirent_unlock(dl);

2151         VN_RELE(vp);

2153         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
```

```
2154                 zil_commit(zilog, 0);

2156         ZFS_EXIT(zfsvfs);
2157         return (error);
2158 }

2160 /*
2161  * Read as many directory entries as will fit into the provided
2162  * buffer from the given directory cursor position (specified in
2163  * the uio structure).
2164  *
2165  *      IN:     vp      - vnode of directory to read.
2166  *              uio     - structure supplying read location, range info,
2167  *                        and return buffer.
2168  *              cr      - credentials of caller.
2169  *              ct      - caller context
2170  *              flags   - case flags
2171  *
2172  *      OUT:    uio     - updated offset and range, buffer filled.
2173  *              eofp    - set to true if end-of-file detected.
2174  *
2175  *      RETURN: 0 on success, error code on failure.
2176  *
2177  * Timestamps:
2178  *      vp - atime updated
2179  *
2180  * Note that the low 4 bits of the cookie returned by zap is always zero.
2181  * This allows us to use the low range for "special" directory entries:
2182  * We use 0 for '.', and 1 for '..'.  If this is the root of the filesystem,
2183  * we use the offset 2 for the '.zfs' directory.
2184  */
2185 /* ARGSUSED */
2186 static int
2187 zfs_readdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
2188     caller_context_t *ct, int flags)
2189 {
2190         znode_t         *zp = VTOZ(vp);
2191         iovec_t         *iovp;
2192         edirent_t       *eodp;
2193         dirent64_t      *odp;
2194         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2195         objset_t        *os;
2196         caddr_t         outbuf;
2197         size_t          bufsize;
2198         zap_cursor_t    zc;
2199         zap_attribute_t zap;
2200         uint_t          bytes_wanted;
2201         uint64_t        offset; /* must be unsigned; checks for < 1 */
2202         uint64_t        parent;
2203         int             local_eof;
2204         int             outcount;
2205         int             error;
2206         uint8_t         prefetch;
2207         boolean_t       check_sysattrs;

2209         ZFS_ENTER(zfsvfs);
2210         ZFS_VERIFY_ZP(zp);

2212         if ((error = sa_lookup(zp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
2213             &parent, sizeof (parent))) != 0) {
2214                 ZFS_EXIT(zfsvfs);
2215                 return (error);
2216         }

2218         /*
2219          * If we are not given an eof variable,
```



```
2220          * use a local one.
2221          */
2222         if (eofp == NULL)
2223                 eofp = &local_eof;

2225         /*
2226          * Check for valid iov_len.
2227          */
2228         if (uio->uio_iov->iov_len <= 0) {
2229                 ZFS_EXIT(zfsvfs);
2230                 return (SET_ERROR(EINVAL));
2231         }

2233         /*
2234          * Quit if directory has been removed (posix)
2235          */
2236         if ((*eofp = zp->z_unlinked) != 0) {
2237                 ZFS_EXIT(zfsvfs);
2238                 return (0);
2239         }

2241         error = 0;
2242         os = zfsvfs->z_os;
2243         offset = uio->uio_loffset;
2244         prefetch = zp->z_zn_prefetch;

2246         /*
2247          * Initialize the iterator cursor.
2248          */
2249         if (offset <= 3) {
2250                 /*
2251                  * Start iteration from the beginning of the directory.
2252                  */
2253                 zap_cursor_init(&zc, os, zp->z_id);
2254         } else {
2255                 /*
2256                  * The offset is a serialized cursor.
2257                  */
2258                 zap_cursor_init_serialized(&zc, os, zp->z_id, offset);
2259         }

2261         /*
2262          * Get space to change directory entries into fs independent format.
2263          */
2264         iovp = uio->uio_iov;
2265         bytes_wanted = iovp->iov_len;
2266         if (uio->uio_segflg != UIO_SYSSPACE || uio->uio_iovcnt != 1) {
2267                 bufsize = bytes_wanted;
2268                 outbuf = kmem_alloc(bufsize, KM_SLEEP);
2269                 odp = (struct dirent64 *)outbuf;
2270         } else {
2271                 bufsize = bytes_wanted;
2272                 outbuf = NULL;
2273                 odp = (struct dirent64 *)iovp->iov_base;
2274         }
2275         eodp = (struct edirent *)odp;

2277         /*
2278          * If this VFS supports the system attribute view interface; and
2279          * we're looking at an extended attribute directory; and we care
2280          * about normalization conflicts on this vfs; then we must check
2281          * for normalization conflicts with the sysattr name space.
2282          */
2283         check_sysattrs = vfs_has_feature(vp->v_vfsp, VFSFT_SYSATTR_VIEWS) &&
2284             (vp->v_flag & V_XATTRDIR) && zfsvfs->z_norm &&
2285             (flags & V_RDDIR_ENTFLAGS);
```

```
2287         /*
2288          * Transform to file-system independent format
2289          */
2290         outcount = 0;
2291         while (outcount < bytes_wanted) {
2292                 ino64_t objnum;
2293                 ushort_t reclen;
2294                 off64_t *next = NULL;

2296                 /*
2297                  * Special case '.', '..', and '.zfs'.
2298                  */
2299                 if (offset == 0) {
2300                         (void) strcpy(zap.za_name, ".");
2301                         zap.za_normalization_conflict = 0;
2302                         objnum = zp->z_id;
2303                 } else if (offset == 1) {
2304                         (void) strcpy(zap.za_name, "..");
2305                         zap.za_normalization_conflict = 0;
2306                         objnum = parent;
2307                 } else if (offset == 2 && zfs_show_ctldir(zp)) {
2308                         (void) strcpy(zap.za_name, ZFS_CTLDIR_NAME);
2309                         zap.za_normalization_conflict = 0;
2310                         objnum = ZFSCTL_INO_ROOT;
2311                 } else {
2312                         /*
2313                          * Grab next entry.
2314                          */
2315                         if (error = zap_cursor_retrieve(&zc, &zap)) {
2316                                 if ((*eofp = (error == ENOENT)) != 0)
2317                                         break;
2318                                 else
2319                                         goto update;
2320                         }

2322                         if (zap.za_integer_length != 8 ||
2323                             zap.za_num_integers != 1) {
2324                                 cmn_err(CE_WARN, "zap_readdir: bad directory "
2325                                     "entry, obj = %lld, offset = %lld\n",
2326                                     (u_longlong_t)zp->z_id,
2327                                     (u_longlong_t)offset);
2328                                 error = SET_ERROR(ENXIO);
2329                                 goto update;
2330                         }

2332                         objnum = ZFS_DIRENT_OBJ(zap.za_first_integer);
2333                         /*
2334                          * MacOS X can extract the object type here such as:
2335                          * uint8_t type = ZFS_DIRENT_TYPE(zap.za_first_integer);
2336                          */

2338                         if (check_sysattrs && !zap.za_normalization_conflict) {
2339                                 zap.za_normalization_conflict =
2340                                     xattr_sysattr_casechk(zap.za_name);
2341                         }
2342                 }

2344                 if (flags & V_RDDIR_ACCFILTER) {
2345                         /*
2346                          * If we have no access at all, don't include
2347                          * this entry in the returned information
2348                          */
2349                         znode_t *ezp;
2350                         if (zfs_zget(zp->z_zfsvfs, objnum, &ezp) != 0)
2351                                 goto skip_entry;
```



```
2352                         if (!zfs_has_access(ezp, cr)) {
2353                                 VN_RELE(ZTOV(ezp));
2354                                 goto skip_entry;
2355                         }
2356                         VN_RELE(ZTOV(ezp));
2357                 }

2359                 if (flags & V_RDDIR_ENTFLAGS)
2360                         reclen = EDIRENT_RECLEN(strlen(zap.za_name));
2361                 else
2362                         reclen = DIRENT64_RECLEN(strlen(zap.za_name));

2364                 /*
2365                  * Will this entry fit in the buffer?
2366                  */
2367                 if (outcount + reclen > bufsize) {
2368                         /*
2369                          * Did we manage to fit anything in the buffer?
2370                          */
2371                         if (!outcount) {
2372                                 error = SET_ERROR(EINVAL);
2373                                 goto update;
2374                         }
2375                         break;
2376                 }
2377                 if (flags & V_RDDIR_ENTFLAGS) {
2378                         /*
2379                          * Add extended flag entry:
2380                          */
2381                         eodp->ed_ino = objnum;
2382                         eodp->ed_reclen = reclen;
2383                         /* NOTE: ed_off is the offset for the *next* entry */
2384                         next = &(eodp->ed_off);
2385                         eodp->ed_eflags = zap.za_normalization_conflict ?
2386                             ED_CASE_CONFLICT : 0;
2387                         (void) strncpy(eodp->ed_name, zap.za_name,
2388                             EDIRENT_NAMELEN(reclen));
2389                         eodp = (edirent_t *)((intptr_t)eodp + reclen);
2390                 } else {
2391                         /*
2392                          * Add normal entry:
2393                          */
2394                         odp->d_ino = objnum;
2395                         odp->d_reclen = reclen;
2396                         /* NOTE: d_off is the offset for the *next* entry */
2397                         next = &(odp->d_off);
2398                         (void) strncpy(odp->d_name, zap.za_name,
2399                             DIRENT64_NAMELEN(reclen));
2400                         odp = (dirent64_t *)((intptr_t)odp + reclen);
2401                 }
2402                 outcount += reclen;

2404                 ASSERT(outcount <= bufsize);

2406                 /* Prefetch znode */
2407                 if (prefetch)
2408                         dmu_prefetch(os, objnum, 0, 0, 0,
2409                             ZIO_PRIORITY_SYNC_READ);

2411         skip_entry:
2412                 /*
2413                  * Move to the next entry, fill in the previous offset.
2414                  */
2415                 if (offset > 2 || (offset == 2 && !zfs_show_ctldir(zp))) {
2416                         zap_cursor_advance(&zc);
2417                         offset = zap_cursor_serialize(&zc);
```

```
2418                 } else {
2419                         offset += 1;
2420                 }
2421                 if (next)
2422                         *next = offset;
2423         }
2424         zp->z_zn_prefetch = B_FALSE; /* a lookup will re-enable pre-fetching */

2426         if (uio->uio_segflg == UIO_SYSSPACE && uio->uio_iovcnt == 1) {
2427                 iovp->iov_base += outcount;
2428                 iovp->iov_len -= outcount;
2429                 uio->uio_resid -= outcount;
2430         } else if (error = uiomove(outbuf, (long)outcount, UIO_READ, uio)) {
2431                 /*
2432                  * Reset the pointer.
2433                  */
2434                 offset = uio->uio_loffset;
2435         }

2437 update:
2438         zap_cursor_fini(&zc);
2439         if (uio->uio_segflg != UIO_SYSSPACE || uio->uio_iovcnt != 1)
2440                 kmem_free(outbuf, bufsize);

2442         if (error == ENOENT)
2443                 error = 0;

2445         ZFS_ACCESSTIME_STAMP(zfsvfs, zp);

2447         uio->uio_loffset = offset;
2448         ZFS_EXIT(zfsvfs);
2449         return (error);
2450 }

2452 ulong_t zfs_fsync_sync_cnt = 4;

2454 static int
2455 zfs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
2456 {
2457         znode_t *zp = VTOZ(vp);
2458         zfsvfs_t *zfsvfs = zp->z_zfsvfs;

2460         /*
2461          * Regardless of whether this is required for standards conformance,
2462          * this is the logical behavior when fsync() is called on a file with
2463          * dirty pages.  We use B_ASYNC since the ZIL transactions are already
2464          * going to be pushed out as part of the zil_commit().
2465          */
2466         if (vn_has_cached_data(vp) && !(syncflag & FNODSYNC) &&
2467             (vp->v_type == VREG) && !(IS_SWAPVP(vp)))
2468                 (void) VOP_PUTPAGE(vp, (offset_t)0, (size_t)0, B_ASYNC, cr, ct);

2470         (void) tsd_set(zfs_fsyncer_key, (void *)zfs_fsync_sync_cnt);

2472         if (zfsvfs->z_os->os_sync != ZFS_SYNC_DISABLED) {
2473                 ZFS_ENTER(zfsvfs);
2474                 ZFS_VERIFY_ZP(zp);
2475                 zil_commit(zfsvfs->z_log, zp->z_id);
2476                 ZFS_EXIT(zfsvfs);
2477         }
2478         return (0);
2479 }


2482 /*
2483  * Get the requested file attributes and place them in the provided
```



```
2484  * vattr structure.
2485  *
2486  *      IN:     vp      - vnode of file.
2487  *              vap     - va_mask identifies requested attributes.
2488  *                        If AT_XVATTR set, then optional attrs are requested
2489  *              flags   - ATTR_NOACLCHECK (CIFS server context)
2490  *              cr      - credentials of caller.
2491  *              ct      - caller context
2492  *
2493  *      OUT:    vap     - attribute values.
2494  *
2495  *      RETURN: 0 (always succeeds).
2496  */
2497 /* ARGSUSED */
2498 static int
2499 zfs_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2500     caller_context_t *ct)
2501 {
2502         znode_t *zp = VTOZ(vp);
2503         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
2504         int     error = 0;
2505         uint64_t links;
2506         uint64_t mtime[2], ctime[2];
2507         xvattr_t *xvap = (xvattr_t *)vap;       /* vap may be an xvattr_t * */
2508         xoptattr_t *xoap = NULL;
2509         boolean_t skipaclchk = (flags & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
2510         sa_bulk_attr_t bulk[2];
2511         int count = 0;

2513         ZFS_ENTER(zfsvfs);
2514         ZFS_VERIFY_ZP(zp);

2516         zfs_fuid_map_ids(zp, cr, &vap->va_uid, &vap->va_gid);

2518         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
2519         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);

2521         if ((error = sa_bulk_lookup(zp->z_sa_hdl, bulk, count)) != 0) {
2522                 ZFS_EXIT(zfsvfs);
2523                 return (error);
2524         }

2526         /*
2527          * If ACL is trivial don't bother looking for ACE_READ_ATTRIBUTES.
2528          * Also, if we are the owner don't bother, since owner should
2529          * always be allowed to read basic attributes of file.
2530          */
2531         if (!(zp->z_pflags & ZFS_ACL_TRIVIAL) &&
2532             (vap->va_uid != crgetuid(cr))) {
2533                 if (error = zfs_zaccess(zp, ACE_READ_ATTRIBUTES, 0,
2534                     skipaclchk, cr)) {
2535                         ZFS_EXIT(zfsvfs);
2536                         return (error);
2537                 }
2538         }

2540         /*
2541          * Return all attributes.  It's cheaper to provide the answer
2542          * than to determine whether we were asked the question.
2543          */

2545         mutex_enter(&zp->z_lock);
2546         vap->va_type = vp->v_type;
2547         vap->va_mode = zp->z_mode & MODEMASK;
2548         vap->va_fsid = zp->z_zfsvfs->z_vfs->vfs_dev;
2549         vap->va_nodeid = zp->z_id;
```

```
2550         if ((vp->v_flag & VROOT) && zfs_show_ctldir(zp))
2551                 links = zp->z_links + 1;
2552         else
2553                 links = zp->z_links;
2554         vap->va_nlink = MIN(links, UINT32_MAX); /* nlink_t limit! */
2555         vap->va_size = zp->z_size;
2556         vap->va_rdev = vp->v_rdev;
2557         vap->va_seq = zp->z_seq;

2559         /*
2560          * Add in any requested optional attributes and the create time.
2561          * Also set the corresponding bits in the returned attribute bitmap.
2562          */
2563         if ((xoap = xva_getxoptattr(xvap)) != NULL && zfsvfs->z_use_fuids) {
2564                 if (XVA_ISSET_REQ(xvap, XAT_ARCHIVE)) {
2565                         xoap->xoa_archive =
2566                             ((zp->z_pflags & ZFS_ARCHIVE) != 0);
2567                         XVA_SET_RTN(xvap, XAT_ARCHIVE);
2568                 }

2570                 if (XVA_ISSET_REQ(xvap, XAT_READONLY)) {
2571                         xoap->xoa_readonly =
2572                             ((zp->z_pflags & ZFS_READONLY) != 0);
2573                         XVA_SET_RTN(xvap, XAT_READONLY);
2574                 }

2576                 if (XVA_ISSET_REQ(xvap, XAT_SYSTEM)) {
2577                         xoap->xoa_system =
2578                             ((zp->z_pflags & ZFS_SYSTEM) != 0);
2579                         XVA_SET_RTN(xvap, XAT_SYSTEM);
2580                 }

2582                 if (XVA_ISSET_REQ(xvap, XAT_HIDDEN)) {
2583                         xoap->xoa_hidden =
2584                             ((zp->z_pflags & ZFS_HIDDEN) != 0);
2585                         XVA_SET_RTN(xvap, XAT_HIDDEN);
2586                 }

2588                 if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2589                         xoap->xoa_nounlink =
2590                             ((zp->z_pflags & ZFS_NOUNLINK) != 0);
2591                         XVA_SET_RTN(xvap, XAT_NOUNLINK);
2592                 }

2594                 if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2595                         xoap->xoa_immutable =
2596                             ((zp->z_pflags & ZFS_IMMUTABLE) != 0);
2597                         XVA_SET_RTN(xvap, XAT_IMMUTABLE);
2598                 }

2600                 if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2601                         xoap->xoa_appendonly =
2602                             ((zp->z_pflags & ZFS_APPENDONLY) != 0);
2603                         XVA_SET_RTN(xvap, XAT_APPENDONLY);
2604                 }

2606                 if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2607                         xoap->xoa_nodump =
2608                             ((zp->z_pflags & ZFS_NODUMP) != 0);
2609                         XVA_SET_RTN(xvap, XAT_NODUMP);
2610                 }

2612                 if (XVA_ISSET_REQ(xvap, XAT_OPAQUE)) {
2613                         xoap->xoa_opaque =
2614                             ((zp->z_pflags & ZFS_OPAQUE) != 0);
2615                         XVA_SET_RTN(xvap, XAT_OPAQUE);
```



```
2616                 }

2618                 if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
2619                         xoap->xoa_av_quarantined =
2620                             ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0);
2621                         XVA_SET_RTN(xvap, XAT_AV_QUARANTINED);
2622                 }

2624                 if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
2625                         xoap->xoa_av_modified =
2626                             ((zp->z_pflags & ZFS_AV_MODIFIED) != 0);
2627                         XVA_SET_RTN(xvap, XAT_AV_MODIFIED);
2628                 }

2630                 if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) &&
2631                     vp->v_type == VREG) {
2632                         zfs_sa_get_scanstamp(zp, xvap);
2633                 }

2635                 if (XVA_ISSET_REQ(xvap, XAT_CREATETIME)) {
2636                         uint64_t times[2];

2638                         (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_CRTIME(zfsvfs),
2639                             times, sizeof (times));
2640                         ZFS_TIME_DECODE(&xoap->xoa_createtime, times);
2641                         XVA_SET_RTN(xvap, XAT_CREATETIME);
2642                 }

2644                 if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
2645                         xoap->xoa_reparse = ((zp->z_pflags & ZFS_REPARSE) != 0);
2646                         XVA_SET_RTN(xvap, XAT_REPARSE);
2647                 }
2648                 if (XVA_ISSET_REQ(xvap, XAT_GEN)) {
2649                         xoap->xoa_generation = zp->z_gen;
2650                         XVA_SET_RTN(xvap, XAT_GEN);
2651                 }

2653                 if (XVA_ISSET_REQ(xvap, XAT_OFFLINE)) {
2654                         xoap->xoa_offline =
2655                             ((zp->z_pflags & ZFS_OFFLINE) != 0);
2656                         XVA_SET_RTN(xvap, XAT_OFFLINE);
2657                 }

2659                 if (XVA_ISSET_REQ(xvap, XAT_SPARSE)) {
2660                         xoap->xoa_sparse =
2661                             ((zp->z_pflags & ZFS_SPARSE) != 0);
2662                         XVA_SET_RTN(xvap, XAT_SPARSE);
2663                 }
2664         }

2666         ZFS_TIME_DECODE(&vap->va_atime, zp->z_atime);
2667         ZFS_TIME_DECODE(&vap->va_mtime, mtime);
2668         ZFS_TIME_DECODE(&vap->va_ctime, ctime);

2670         mutex_exit(&zp->z_lock);

2672         sa_object_size(zp->z_sa_hdl, &vap->va_blksize, &vap->va_nblocks);

2674         if (zp->z_blksz == 0) {
2675                 /*
2676                  * Block size hasn't been set; suggest maximal I/O transfers.
2677                  */
2678                 vap->va_blksize = zfsvfs->z_max_blksz;
2679         }

2681         ZFS_EXIT(zfsvfs);
```

```
2682         return (0);
2683 }

2685 /*
2686  * Set the file attributes to the values contained in the
2687  * vattr structure.
2688  *
2689  *      IN:     vp      - vnode of file to be modified.
2690  *              vap     - new attribute values.
2691  *                        If AT_XVATTR set, then optional attrs are being set
2692  *              flags   - ATTR_UTIME set if non-default time values provided.
2693  *                      - ATTR_NOACLCHECK (CIFS context only).
2694  *              cr      - credentials of caller.
2695  *              ct      - caller context
2696  *
2697  *      RETURN: 0 on success, error code on failure.
2698  *
2699  * Timestamps:
2700  *      vp - ctime updated, mtime updated if size changed.
2701  */
2702 /* ARGSUSED */
2703 static int
2704 zfs_setattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2705     caller_context_t *ct)
2706 {
2707         znode_t         *zp = VTOZ(vp);
2708         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2709         zilog_t         *zilog;
2710         dmu_tx_t        *tx;
2711         vattr_t         oldva;
2712         xvattr_t        tmpxvattr;
2713         uint_t          mask = vap->va_mask;
2714         uint_t          saved_mask = 0;
2715         int             trim_mask = 0;
2716         uint64_t        new_mode;
2717         uint64_t        new_uid, new_gid;
2718         uint64_t        xattr_obj;
2719         uint64_t        mtime[2], ctime[2];
2720         znode_t         *attrzp;
2721         int             need_policy = FALSE;
2722         int             err, err2;
2723         zfs_fuid_info_t *fuidp = NULL;
2724         xvattr_t *xvap = (xvattr_t *)vap;       /* vap may be an xvattr_t * */
2725         xoptattr_t      *xoap;
2726         zfs_acl_t       *aclp;
2727         boolean_t skipaclchk = (flags & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
2728         boolean_t       fuid_dirtied = B_FALSE;
2729         sa_bulk_attr_t  bulk[7], xattr_bulk[7];
2730         int             count = 0, xattr_count = 0;

2732         if (mask == 0)
2733                 return (0);

2735         if (mask & AT_NOSET)
2736                 return (SET_ERROR(EINVAL));

2738         ZFS_ENTER(zfsvfs);
2739         ZFS_VERIFY_ZP(zp);

2741         zilog = zfsvfs->z_log;

2743         /*
2744          * Make sure that if we have ephemeral uid/gid or xvattr specified
2745          * that file system is at proper version level
2746          */
```



```
2748         if (zfsvfs->z_use_fuids == B_FALSE &&
2749             (((mask & AT_UID) && IS_EPHEMERAL(vap->va_uid)) ||
2750             ((mask & AT_GID) && IS_EPHEMERAL(vap->va_gid)) ||
2751             (mask & AT_XVATTR))) {
2752                 ZFS_EXIT(zfsvfs);
2753                 return (SET_ERROR(EINVAL));
2754         }

2756         if (mask & AT_SIZE && vp->v_type == VDIR) {
2757                 ZFS_EXIT(zfsvfs);
2758                 return (SET_ERROR(EISDIR));
2759         }

2761         if (mask & AT_SIZE && vp->v_type != VREG && vp->v_type != VFIFO) {
2762                 ZFS_EXIT(zfsvfs);
2763                 return (SET_ERROR(EINVAL));
2764         }

2766         /*
2767          * If this is an xvattr_t, then get a pointer to the structure of
2768          * optional attributes.  If this is NULL, then we have a vattr_t.
2769          */
2770         xoap = xva_getxoptattr(xvap);

2772         xva_init(&tmpxvattr);

2774         /*
2775          * Immutable files can only alter immutable bit and atime
2776          */
2777         if ((zp->z_pflags & ZFS_IMMUTABLE) &&
2778             ((mask & (AT_SIZE|AT_UID|AT_GID|AT_MTIME|AT_MODE)) ||
2779             ((mask & AT_XVATTR) && XVA_ISSET_REQ(xvap, XAT_CREATETIME)))) {
2780                 ZFS_EXIT(zfsvfs);
2781                 return (SET_ERROR(EPERM));
2782         }

2784         if ((mask & AT_SIZE) && (zp->z_pflags & ZFS_READONLY)) {
2785                 ZFS_EXIT(zfsvfs);
2786                 return (SET_ERROR(EPERM));
2787         }

2789         /*
2790          * Verify timestamps doesn't overflow 32 bits.
2791          * ZFS can handle large timestamps, but 32bit syscalls can't
2792          * handle times greater than 2039.  This check should be removed
2793          * once large timestamps are fully supported.
2794          */
2795         if (mask & (AT_ATIME | AT_MTIME)) {
2796                 if (((mask & AT_ATIME) && TIMESPEC_OVERFLOW(&vap->va_atime)) ||
2797                     ((mask & AT_MTIME) && TIMESPEC_OVERFLOW(&vap->va_mtime))) {
2798                         ZFS_EXIT(zfsvfs);
2799                         return (SET_ERROR(EOVERFLOW));
2800                 }
2801         }

2803 top:
2804         attrzp = NULL;
2805         aclp = NULL;

2807         /* Can this be moved to before the top label? */
2808         if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
2809                 ZFS_EXIT(zfsvfs);
2810                 return (SET_ERROR(EROFS));
2811         }

2813         /*
```

```
2814          * First validate permissions
2815          */
2817         if (mask & AT_SIZE) {
2818                 err = zfs_zaccess(zp, ACE_WRITE_DATA, 0, skipaclchk, cr);
2819                 if (err) {
2820                         ZFS_EXIT(zfsvfs);
2821                         return (err);
2822                 }
2823                 /*
2824                  * XXX - Note, we are not providing any open
2825                  * mode flags here (like FNDELAY), so we may
2826                  * block if there are locks present... this
2827                  * should be addressed in openat().
2828                  */
2829                 /* XXX - would it be OK to generate a log record here? */
2830                 err = zfs_freesp(zp, vap->va_size, 0, 0, FALSE);
2831                 if (err) {
2832                         ZFS_EXIT(zfsvfs);
2833                         return (err);
2834                 }
2836                 if (vap->va_size == 0)
2837                         vnevent_truncate(ZTOV(zp), ct);
2838         }
2840         if (mask & (AT_ATIME|AT_MTIME) ||
2841             ((mask & AT_XVATTR) && (XVA_ISSET_REQ(xvap, XAT_HIDDEN) ||
2842             XVA_ISSET_REQ(xvap, XAT_READONLY) ||
2843             XVA_ISSET_REQ(xvap, XAT_ARCHIVE) ||
2844             XVA_ISSET_REQ(xvap, XAT_OFFLINE) ||
2845             XVA_ISSET_REQ(xvap, XAT_SPARSE) ||
2846             XVA_ISSET_REQ(xvap, XAT_CREATETIME) ||
2847             XVA_ISSET_REQ(xvap, XAT_SYSTEM)))) {
2848                 need_policy = zfs_zaccess(zp, ACE_WRITE_ATTRIBUTES, 0,
2849                     skipaclchk, cr);
2850         }
2852         if (mask & (AT_UID|AT_GID)) {
2853                 int     idmask = (mask & (AT_UID|AT_GID));
2854                 int     take_owner;
2855                 int     take_group;
2857                 /*
2858                  * NOTE: even if a new mode is being set,
2859                  * we may clear S_ISUID/S_ISGID bits.
2860                  */
2862                 if (!(mask & AT_MODE))
2863                         vap->va_mode = zp->z_mode;
2865                 /*
2866                  * Take ownership or chgrp to group we are a member of
2867                  */
2869                 take_owner = (mask & AT_UID) && (vap->va_uid == crgetuid(cr));
2870                 take_group = (mask & AT_GID) &&
2871                     zfs_groupmember(zfsvfs, vap->va_gid, cr);
2873                 /*
2874                  * If both AT_UID and AT_GID are set then take_owner and
2875                  * take_group must both be set in order to allow taking
2876                  * ownership.
2877                  *
2878                  * Otherwise, send the check through secpolicy_vnode_setattr()
2879                  *
```



```
2880                  */
2882                 if (((idmask == (AT_UID|AT_GID)) && take_owner && take_group) ||
2883                     ((idmask == AT_UID) && take_owner) ||
2884                     ((idmask == AT_GID) && take_group)) {
2885                         if (zfs_zaccess(zp, ACE_WRITE_OWNER, 0,
2886                             skipaclchk, cr) == 0) {
2887                                 /*
2888                                  * Remove setuid/setgid for non-privileged users
2889                                  */
2890                                 secpolicy_setid_clear(vap, cr);
2891                                 trim_mask = (mask & (AT_UID|AT_GID));
2892                         } else {
2893                                 need_policy =  TRUE;
2894                         }
2895                 } else {
2896                         need_policy =  TRUE;
2897                 }
2898         }
2900         mutex_enter(&zp->z_lock);
2901         oldva.va_mode = zp->z_mode;
2902         zfs_fuid_map_ids(zp, cr, &oldva.va_uid, &oldva.va_gid);
2903         if (mask & AT_XVATTR) {
2904                 /*
2905                  * Update xvattr mask to include only those attributes
2906                  * that are actually changing.
2907                  *
2908                  * the bits will be restored prior to actually setting
2909                  * the attributes so the caller thinks they were set.
2910                  */
2911                 if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2912                         if (xoap->xoa_appendonly !=
2913                             ((zp->z_pflags & ZFS_APPENDONLY) != 0)) {
2914                                 need_policy = TRUE;
2915                         } else {
2916                                 XVA_CLR_REQ(xvap, XAT_APPENDONLY);
2917                                 XVA_SET_REQ(&tmpxvattr, XAT_APPENDONLY);
2918                         }
2919                 }
2921                 if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2922                         if (xoap->xoa_nounlink !=
2923                             ((zp->z_pflags & ZFS_NOUNLINK) != 0)) {
2924                                 need_policy = TRUE;
2925                         } else {
2926                                 XVA_CLR_REQ(xvap, XAT_NOUNLINK);
2927                                 XVA_SET_REQ(&tmpxvattr, XAT_NOUNLINK);
2928                         }
2929                 }
2931                 if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2932                         if (xoap->xoa_immutable !=
2933                             ((zp->z_pflags & ZFS_IMMUTABLE) != 0)) {
2934                                 need_policy = TRUE;
2935                         } else {
2936                                 XVA_CLR_REQ(xvap, XAT_IMMUTABLE);
2937                                 XVA_SET_REQ(&tmpxvattr, XAT_IMMUTABLE);
2938                         }
2939                 }
2941                 if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2942                         if (xoap->xoa_nodump !=
2943                             ((zp->z_pflags & ZFS_NODUMP) != 0)) {
2944                                 need_policy = TRUE;
2945                         } else {
```

```
2946                                XVA_CLR_REQ(xvap, XAT_NODUMP);
2947                                XVA_SET_REQ(&tmpxvattr, XAT_NODUMP);
2948                        }
2949                }

2951                if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
2952                        if (xoap->xoa_av_modified !=
2953                            ((zp->z_pflags & ZFS_AV_MODIFIED) != 0)) {
2954                                need_policy = TRUE;
2955                        } else {
2956                                XVA_CLR_REQ(xvap, XAT_AV_MODIFIED);
2957                                XVA_SET_REQ(&tmpxvattr, XAT_AV_MODIFIED);
2958                        }
2959                }

2961                if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
2962                        if ((vp->v_type != VREG &&
2963                            xoap->xoa_av_quarantined) ||
2964                            xoap->xoa_av_quarantined !=
2965                            ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0)) {
2966                                need_policy = TRUE;
2967                        } else {
2968                                XVA_CLR_REQ(xvap, XAT_AV_QUARANTINED);
2969                                XVA_SET_REQ(&tmpxvattr, XAT_AV_QUARANTINED);
2970                        }
2971                }

2973                if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
2974                        mutex_exit(&zp->z_lock);
2975                        ZFS_EXIT(zfsvfs);
2976                        return (SET_ERROR(EPERM));
2977                }

2979                if (need_policy == FALSE &&
2980                    (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) ||
2981                    XVA_ISSET_REQ(xvap, XAT_OPAQUE))) {
2982                        need_policy = TRUE;
2983                }
2984        }

2986        mutex_exit(&zp->z_lock);

2988        if (mask & AT_MODE) {
2989                if (zfs_zaccess(zp, ACE_WRITE_ACL, 0, skipaclchk, cr) == 0) {
2990                        err = secpolicy_setid_setsticky_clear(vp, vap,
2991                            &oldva, cr);
2992                        if (err) {
2993                                ZFS_EXIT(zfsvfs);
2994                                return (err);
2995                        }
2996                        trim_mask |= AT_MODE;
2997                } else {
2998                        need_policy = TRUE;
2999                }
3000        }

3002        if (need_policy) {
3003                /*
3004                 * If trim_mask is set then take ownership
3005                 * has been granted or write_acl is present and user
3006                 * has the ability to modify mode.  In that case remove
3007                 * UID|GID and or MODE from mask so that
3008                 * secpolicy_vnode_setattr() doesn't revoke it.
3009                 */

3011                if (trim_mask) {
```



```
3012                        saved_mask = vap->va_mask;
3013                        vap->va_mask &= ~trim_mask;
3014                }
3015                err = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
3016                    (int (*)(void *, int, cred_t *))zfs_zaccess_unix, zp);
3017                if (err) {
3018                        ZFS_EXIT(zfsvfs);
3019                        return (err);
3020                }

3022                if (trim_mask)
3023                        vap->va_mask |= saved_mask;
3024        }

3026        /*
3027         * secpolicy_vnode_setattr, or take ownership may have
3028         * changed va_mask
3029         */
3030        mask = vap->va_mask;

3032        if ((mask & (AT_UID | AT_GID))) {
3033                err = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
3034                    &xattr_obj, sizeof (xattr_obj));

3036                if (err == 0 && xattr_obj) {
3037                        err = zfs_zget(zp->z_zfsvfs, xattr_obj, &attrzp);
3038                        if (err)
3039                                goto out2;
3040                }
3041                if (mask & AT_UID) {
3042                        new_uid = zfs_fuid_create(zfsvfs,
3043                            (uint64_t)vap->va_uid, cr, ZFS_OWNER, &fuidp);
3044                        if (new_uid != zp->z_uid &&
3045                            zfs_fuid_overquota(zfsvfs, B_FALSE, new_uid)) {
3046                                if (attrzp)
3047                                        VN_RELE(ZTOV(attrzp));
3048                                err = SET_ERROR(EDQUOT);
3049                                goto out2;
3050                        }
3051                }

3053                if (mask & AT_GID) {
3054                        new_gid = zfs_fuid_create(zfsvfs, (uint64_t)vap->va_gid,
3055                            cr, ZFS_GROUP, &fuidp);
3056                        if (new_gid != zp->z_gid &&
3057                            zfs_fuid_overquota(zfsvfs, B_TRUE, new_gid)) {
3058                                if (attrzp)
3059                                        VN_RELE(ZTOV(attrzp));
3060                                err = SET_ERROR(EDQUOT);
3061                                goto out2;
3062                        }
3063                }
3064        }
3065        tx = dmu_tx_create(zfsvfs->z_os);

3067        if (mask & AT_MODE) {
3068                uint64_t pmode = zp->z_mode;
3069                uint64_t acl_obj;
3070                new_mode = (pmode & S_IFMT) | (vap->va_mode & ~S_IFMT);

3072                if (zp->z_zfsvfs->z_acl_mode == ZFS_ACL_RESTRICTED &&
3073                    !(zp->z_pflags & ZFS_ACL_TRIVIAL)) {
3074                        err = SET_ERROR(EPERM);
3075                        goto out;
3076                }
```

```
3078                if (err = zfs_acl_chmod_setattr(zp, &aclp, new_mode))
3079                        goto out;

3081                mutex_enter(&zp->z_lock);
3082                if (!zp->z_is_sa && ((acl_obj = zfs_external_acl(zp)) != 0)) {
3083                        /*
3084                         * Are we upgrading ACL from old V0 format
3085                         * to V1 format?
3086                         */
3087                        if (zfsvfs->z_version >= ZPL_VERSION_FUID &&
3088                            zfs_znode_acl_version(zp) ==
3089                            ZFS_ACL_VERSION_INITIAL) {
3090                                dmu_tx_hold_free(tx, acl_obj, 0,
3091                                    DMU_OBJECT_END);
3092                                dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3093                                    0, aclp->z_acl_bytes);
3094                        } else {
3095                                dmu_tx_hold_write(tx, acl_obj, 0,
3096                                    aclp->z_acl_bytes);
3097                        }
3098                } else if (!zp->z_is_sa && aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3099                        dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3100                            0, aclp->z_acl_bytes);
3101                }
3102                mutex_exit(&zp->z_lock);
3103                dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3104        } else {
3105                if ((mask & AT_XVATTR) &&
3106                    XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3107                        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3108                else
3109                        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
3110        }

3112        if (attrzp) {
3113                dmu_tx_hold_sa(tx, attrzp->z_sa_hdl, B_FALSE);
3114        }

3116        fuid_dirtied = zfsvfs->z_fuid_dirty;
3117        if (fuid_dirtied)
3118                zfs_fuid_txhold(zfsvfs, tx);

3120        zfs_sa_upgrade_txholds(tx, zp);

3122        err = dmu_tx_assign(tx, TXG_WAIT);
3123        if (err)
3124                goto out;

3126        count = 0;
3127        /*
3128         * Set each attribute requested.
3129         * We group settings according to the locks they need to acquire.
3130         *
3131         * Note: you cannot set ctime directly, although it will be
3132         * updated as a side-effect of calling this function.
3133         */


3136        if (mask & (AT_UID|AT_GID|AT_MODE))
3137                mutex_enter(&zp->z_acl_lock);
3138        mutex_enter(&zp->z_lock);

3140        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
3141            &zp->z_pflags, sizeof (zp->z_pflags));

3143        if (attrzp) {
```



```
3144                if (mask & (AT_UID|AT_GID|AT_MODE))
3145                        mutex_enter(&attrzp->z_acl_lock);
3146                mutex_enter(&attrzp->z_lock);
3147                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3148                    SA_ZPL_FLAGS(zfsvfs), NULL, &attrzp->z_pflags,
3149                    sizeof (attrzp->z_pflags));
3150        }

3152        if (mask & (AT_UID|AT_GID)) {

3154                if (mask & AT_UID) {
3155                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_UID(zfsvfs), NULL,
3156                            &new_uid, sizeof (new_uid));
3157                        zp->z_uid = new_uid;
3158                        if (attrzp) {
3159                                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3160                                    SA_ZPL_UID(zfsvfs), NULL, &new_uid,
3161                                    sizeof (new_uid));
3162                                attrzp->z_uid = new_uid;
3163                        }
3164                }

3166                if (mask & AT_GID) {
3167                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_GID(zfsvfs),
3168                            NULL, &new_gid, sizeof (new_gid));
3169                        zp->z_gid = new_gid;
3170                        if (attrzp) {
3171                                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3172                                    SA_ZPL_GID(zfsvfs), NULL, &new_gid,
3173                                    sizeof (new_gid));
3174                                attrzp->z_gid = new_gid;
3175                        }
3176                }
3177                if (!(mask & AT_MODE)) {
3178                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs),
3179                            NULL, &new_mode, sizeof (new_mode));
3180                        new_mode = zp->z_mode;
3181                }
3182                err = zfs_acl_chown_setattr(zp);
3183                ASSERT(err == 0);
3184                if (attrzp) {
3185                        err = zfs_acl_chown_setattr(attrzp);
3186                        ASSERT(err == 0);
3187                }
3188        }

3190        if (mask & AT_MODE) {
3191                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs), NULL,
3192                    &new_mode, sizeof (new_mode));
3193                zp->z_mode = new_mode;
3194                ASSERT3U((uintptr_t)aclp, !=, NULL);
3195                err = zfs_aclset_common(zp, aclp, cr, tx);
3196                ASSERT0(err);
3197                if (zp->z_acl_cached)
3198                        zfs_acl_free(zp->z_acl_cached);
3199                zp->z_acl_cached = aclp;
3200                aclp = NULL;
3201        }


3204        if (mask & AT_ATIME) {
3205                ZFS_TIME_ENCODE(&vap->va_atime, zp->z_atime);
3206                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ATIME(zfsvfs), NULL,
3207                    &zp->z_atime, sizeof (zp->z_atime));
3208        }
```

```
3210         if (mask & AT_MTIME) {
3211                 ZFS_TIME_ENCODE(&vap->va_mtime, mtime);
3212                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
3213                     mtime, sizeof (mtime));
3214         }

3216         /* XXX - shouldn't this be done *before* the ATIME/MTIME checks? */
3217         if (mask & AT_SIZE && !(mask & AT_MTIME)) {
3218                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs),
3219                     NULL, mtime, sizeof (mtime));
3220                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3221                     &ctime, sizeof (ctime));
3222                 zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
3223                     B_TRUE);
3224         } else if (mask != 0) {
3225                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3226                     &ctime, sizeof (ctime));
3227                 zfs_tstamp_update_setup(zp, STATE_CHANGED, mtime, ctime,
3228                     B_TRUE);
3229                 if (attrzp) {
3230                         SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3231                             SA_ZPL_CTIME(zfsvfs), NULL,
3232                             &ctime, sizeof (ctime));
3233                         zfs_tstamp_update_setup(attrzp, STATE_CHANGED,
3234                             mtime, ctime, B_TRUE);
3235                 }
3236         }
3237         /*
3238          * Do this after setting timestamps to prevent timestamp
3239          * update from toggling bit
3240          */

3242         if (xoap && (mask & AT_XVATTR)) {

3244                 /*
3245                  * restore trimmed off masks
3246                  * so that return masks can be set for caller.
3247                  */

3249                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_APPENDONLY)) {
3250                         XVA_SET_REQ(xvap, XAT_APPENDONLY);
3251                 }
3252                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_NOUNLINK)) {
3253                         XVA_SET_REQ(xvap, XAT_NOUNLINK);
3254                 }
3255                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_IMMUTABLE)) {
3256                         XVA_SET_REQ(xvap, XAT_IMMUTABLE);
3257                 }
3258                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_NODUMP)) {
3259                         XVA_SET_REQ(xvap, XAT_NODUMP);
3260                 }
3261                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_MODIFIED)) {
3262                         XVA_SET_REQ(xvap, XAT_AV_MODIFIED);
3263                 }
3264                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_QUARANTINED)) {
3265                         XVA_SET_REQ(xvap, XAT_AV_QUARANTINED);
3266                 }

3268                 if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3269                         ASSERT(vp->v_type == VREG);

3271                 zfs_xvattr_set(zp, xvap, tx);
3272         }

3274         if (fuid_dirtied)
3275                 zfs_fuid_sync(zfsvfs, tx);
```



```
3277         if (mask != 0)
3278                 zfs_log_setattr(zilog, tx, TX_SETATTR, zp, vap, mask, fuidp);

3280         mutex_exit(&zp->z_lock);
3281         if (mask & (AT_UID|AT_GID|AT_MODE))
3282                 mutex_exit(&zp->z_acl_lock);

3284         if (attrzp) {
3285                 if (mask & (AT_UID|AT_GID|AT_MODE))
3286                         mutex_exit(&attrzp->z_acl_lock);
3287                 mutex_exit(&attrzp->z_lock);
3288         }
3289 out:
3290         if (err == 0 && attrzp) {
3291                 err2 = sa_bulk_update(attrzp->z_sa_hdl, xattr_bulk,
3292                     xattr_count, tx);
3293                 ASSERT(err2 == 0);
3294         }

3296         if (attrzp)
3297                 VN_RELE(ZTOV(attrzp));

3299         if (aclp)
3300                 zfs_acl_free(aclp);

3302         if (fuidp) {
3303                 zfs_fuid_info_free(fuidp);
3304                 fuidp = NULL;
3305         }

3307         if (err) {
3308                 dmu_tx_abort(tx);
3309                 if (err == ERESTART)
3310                         goto top;
3311         } else {
3312                 err2 = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
3313                 dmu_tx_commit(tx);
3314         }

3316 out2:
3317         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3318                 zil_commit(zilog, 0);

3320         ZFS_EXIT(zfsvfs);
3321         return (err);
3322 }

3324 typedef struct zfs_zlock {
3325         krwlock_t       *zl_rwlock;     /* lock we acquired */
3326         znode_t         *zl_znode;      /* znode we held */
3327         struct zfs_zlock *zl_next;      /* next in list */
3328 } zfs_zlock_t;

3330 /*
3331  * Drop locks and release vnodes that were held by zfs_rename_lock().
3332  */
3333 static void
3334 zfs_rename_unlock(zfs_zlock_t **zlpp)
3335 {
3336         zfs_zlock_t *zl;

3338         while ((zl = *zlpp) != NULL) {
3339                 if (zl->zl_znode != NULL)
3340                         VN_RELE(ZTOV(zl->zl_znode));
3341                 rw_exit(zl->zl_rwlock);
```

```
3342                *zlpp = zl->zl_next;
3343                kmem_free(zl, sizeof (*zl));
3344        }
3345 }

3347 /*
3348  * Search back through the directory tree, using the ".." entries.
3349  * Lock each directory in the chain to prevent concurrent renames.
3350  * Fail any attempt to move a directory into one of its own descendants.
3351  * XXX - z_parent_lock can overlap with map or grow locks
3352  */
3353 static int
3354 zfs_rename_lock(znode_t *szp, znode_t *tdzp, znode_t *sdzp, zfs_zlock_t **zlpp)
3355 {
3356        zfs_zlock_t     *zl;
3357        znode_t         *zp = tdzp;
3358        uint64_t        rootid = zp->z_zfsvfs->z_root;
3359        uint64_t        oidp = zp->z_id;
3360        krwlock_t       *rwlp = &szp->z_parent_lock;
3361        krw_t           rw = RW_WRITER;

3363        /*
3364         * First pass write-locks szp and compares to zp->z_id.
3365         * Later passes read-lock zp and compare to zp->z_parent.
3366         */
3367        do {
3368                if (!rw_tryenter(rwlp, rw)) {
3369                        /*
3370                         * Another thread is renaming in this path.
3371                         * Note that if we are a WRITER, we don't have any
3372                         * parent_locks held yet.
3373                         */
3374                        if (rw == RW_READER && zp->z_id > szp->z_id) {
3375                                /*
3376                                 * Drop our locks and restart
3377                                 */
3378                                zfs_rename_unlock(&zl);
3379                                *zlpp = NULL;
3380                                zp = tdzp;
3381                                oidp = zp->z_id;
3382                                rwlp = &szp->z_parent_lock;
3383                                rw = RW_WRITER;
3384                                continue;
3385                        } else {
3386                                /*
3387                                 * Wait for other thread to drop its locks
3388                                 */
3389                                rw_enter(rwlp, rw);
3390                        }
3391                }

3393                zl = kmem_alloc(sizeof (*zl), KM_SLEEP);
3394                zl->zl_rwlock = rwlp;
3395                zl->zl_znode = NULL;
3396                zl->zl_next = *zlpp;
3397                *zlpp = zl;

3399                if (oidp == szp->z_id)          /* We're a descendant of szp */
3400                        return (SET_ERROR(EINVAL));

3402                if (oidp == rootid)             /* We've hit the top */
3403                        return (0);

3405                if (rw == RW_READER) {          /* i.e. not the first pass */
3406                        int error = zfs_zget(zp->z_zfsvfs, oidp, &zp);
3407                        if (error)
```



```
3408                                return (error);
3409                        zl->zl_znode = zp;
3410                }
3411                (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_PARENT(zp->z_zfsvfs),
3412                    &oidp, sizeof (oidp));
3413                rwlp = &zp->z_parent_lock;
3414                rw = RW_READER;

3416        } while (zp->z_id != sdzp->z_id);

3418        return (0);
3419 }

3421 /*
3422  * Move an entry from the provided source directory to the target
3423  * directory.  Change the entry name as indicated.
3424  *
3425  *      IN:     sdvp    - Source directory containing the "old entry".
3426  *              snm     - Old entry name.
3427  *              tdvp    - Target directory to contain the "new entry".
3428  *              tnm     - New entry name.
3429  *              cr      - credentials of caller.
3430  *              ct      - caller context
3431  *              flags   - case flags
3432  *
3433  *      RETURN: 0 on success, error code on failure.
3434  *
3435  * Timestamps:
3436  *      sdvp,tdvp - ctime|mtime updated
3437  */
3438 /*ARGSUSED*/
3439 static int
3440 zfs_rename(vnode_t *sdvp, char *snm, vnode_t *tdvp, char *tnm, cred_t *cr,
3441     caller_context_t *ct, int flags)
3442 {
3443        znode_t         *tdzp, *szp, *tzp;
3444        znode_t         *sdzp = VTOZ(sdvp);
3445        zfsvfs_t        *zfsvfs = sdzp->z_zfsvfs;
3446        zilog_t         *zilog;
3447        vnode_t         *realvp;
3448        zfs_dirlock_t   *sdl, *tdl;
3449        dmu_tx_t        *tx;
3450        zfs_zlock_t     *zl;
3451        int             cmp, serr, terr;
3452        int             error = 0, rm_err = 0;
3453        int             zflg = 0;
3454        boolean_t       waited = B_FALSE;

3456        ZFS_ENTER(zfsvfs);
3457        ZFS_VERIFY_ZP(sdzp);
3458        zilog = zfsvfs->z_log;

3460        /*
3461         * Make sure we have the real vp for the target directory.
3462         */
3463        if (VOP_REALVP(tdvp, &realvp, ct) == 0)
3464                tdvp = realvp;

3466        tdzp = VTOZ(tdvp);
3467        ZFS_VERIFY_ZP(tdzp);

3469        /*
3470         * We check z_zfsvfs rather than v_vfsp here, because snapshots and the
3471         * ctldir appear to have the same v_vfsp.
3472         */
3473        if (tdzp->z_zfsvfs != zfsvfs || zfsctl_is_node(tdvp)) {
```

```
3474                 ZFS_EXIT(zfsvfs);
3475                 return (SET_ERROR(EXDEV));
3476         }

3478         if (zfsvfs->z_utf8 && u8_validate(tnm,
3479             strlen(tnm), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3480                 ZFS_EXIT(zfsvfs);
3481                 return (SET_ERROR(EILSEQ));
3482         }

3484         if (flags & FIGNORECASE)
3485                 zflg |= ZCILOOK;

3487 top:
3488         szp = NULL;
3489         tzp = NULL;
3490         zl = NULL;

3492         /*
3493          * This is to prevent the creation of links into attribute space
3494          * by renaming a linked file into/outof an attribute directory.
3495          * See the comment in zfs_link() for why this is considered bad.
3496          */
3497         if ((tdzp->z_pflags & ZFS_XATTR) != (sdzp->z_pflags & ZFS_XATTR)) {
3498                 ZFS_EXIT(zfsvfs);
3499                 return (SET_ERROR(EINVAL));
3500         }

3502         /*
3503          * Lock source and target directory entries.  To prevent deadlock,
3504          * a lock ordering must be defined.  We lock the directory with
3505          * the smallest object id first, or if it's a tie, the one with
3506          * the lexically first name.
3507          */
3508         if (sdzp->z_id < tdzp->z_id) {
3509                 cmp = -1;
3510         } else if (sdzp->z_id > tdzp->z_id) {
3511                 cmp = 1;
3512         } else {
3513                 /*
3514                  * First compare the two name arguments without
3515                  * considering any case folding.
3516                  */
3517                 int nofold = (zfsvfs->z_norm & ~U8_TEXTPREP_TOUPPER);

3519                 cmp = u8_strcmp(snm, tnm, 0, nofold, U8_UNICODE_LATEST, &error);
3520                 ASSERT(error == 0 || !zfsvfs->z_utf8);
3521                 if (cmp == 0) {
3522                         /*
3523                          * POSIX: "If the old argument and the new argument
3524                          * both refer to links to the same existing file,
3525                          * the rename() function shall return successfully
3526                          * and perform no other action."
3527                          */
3528                         ZFS_EXIT(zfsvfs);
3529                         return (0);
3530                 }
3531                 /*
3532                  * If the file system is case-folding, then we may
3533                  * have some more checking to do.  A case-folding file
3534                  * system is either supporting mixed case sensitivity
3535                  * access or is completely case-insensitive.  Note
3536                  * that the file system is always case preserving.
3537                  *
3538                  * In mixed sensitivity mode case sensitive behavior
3539                  * is the default.  FIGNORECASE must be used to
```



```
3540                  * explicitly request case insensitive behavior.
3541                  *
3542                  * If the source and target names provided differ only
3543                  * by case (e.g., a request to rename 'tim' to 'Tim'),
3544                  * we will treat this as a special case in the
3545                  * case-insensitive mode: as long as the source name
3546                  * is an exact match, we will allow this to proceed as
3547                  * a name-change request.
3548                  */
3549                 if ((zfsvfs->z_case == ZFS_CASE_INSENSITIVE ||
3550                     (zfsvfs->z_case == ZFS_CASE_MIXED &&
3551                     flags & FIGNORECASE)) &&
3552                     u8_strcmp(snm, tnm, 0, zfsvfs->z_norm, U8_UNICODE_LATEST,
3553                     &error) == 0) {
3554                         /*
3555                          * case preserving rename request, require exact
3556                          * name matches
3557                          */
3558                         zflg |= ZCIEXACT;
3559                         zflg &= ~ZCILOOK;
3560                 }
3561         }

3563         /*
3564          * If the source and destination directories are the same, we should
3565          * grab the z_name_lock of that directory only once.
3566          */
3567         if (sdzp == tdzp) {
3568                 zflg |= ZHAVELOCK;
3569                 rw_enter(&sdzp->z_name_lock, RW_READER);
3570         }

3572         if (cmp < 0) {
3573                 serr = zfs_dirent_lock(&sdl, sdzp, snm, &szp,
3574                     ZEXISTS | zflg, NULL, NULL);
3575                 terr = zfs_dirent_lock(&tdl,
3576                     tdzp, tnm, &tzp, ZRENAMING | zflg, NULL, NULL);
3577         } else {
3578                 terr = zfs_dirent_lock(&tdl,
3579                     tdzp, tnm, &tzp, zflg, NULL, NULL);
3580                 serr = zfs_dirent_lock(&sdl,
3581                     sdzp, snm, &szp, ZEXISTS | ZRENAMING | zflg,
3582                     NULL, NULL);
3583         }

3585         if (serr) {
3586                 /*
3587                  * Source entry invalid or not there.
3588                  */
3589                 if (!terr) {
3590                         zfs_dirent_unlock(tdl);
3591                         if (tzp)
3592                                 VN_RELE(ZTOV(tzp));
3593                 }

3595                 if (sdzp == tdzp)
3596                         rw_exit(&sdzp->z_name_lock);

3598                 if (strcmp(snm, "..") == 0)
3599                         serr = SET_ERROR(EINVAL);
3600                 ZFS_EXIT(zfsvfs);
3601                 return (serr);
3602         }
3603         if (terr) {
3604                 zfs_dirent_unlock(sdl);
3605                 VN_RELE(ZTOV(szp));
```

```
3607                if (sdzp == tdzp)
3608                        rw_exit(&sdzp->z_name_lock);
3609
3610                if (strcmp(tnm, "..") == 0)
3611                        terr = SET_ERROR(EINVAL);
3612                ZFS_EXIT(zfsvfs);
3613                return (terr);
3614        }
3615
3616        /*
3617         * Must have write access at the source to remove the old entry
3618         * and write access at the target to create the new entry.
3619         * Note that if target and source are the same, this can be
3620         * done in a single check.
3621         */
3622
3623        if (error = zfs_zaccess_rename(sdzp, szp, tdzp, tzp, cr))
3624                goto out;
3625
3626        if (ZTOV(szp)->v_type == VDIR) {
3627                /*
3628                 * Check to make sure rename is valid.
3629                 * Can’t do a move like this: /usr/a/b to /usr/a/b/c/d
3630                 */
3631                if (error = zfs_rename_lock(szp, tdzp, sdzp, &zl))
3632                        goto out;
3633        }
3634
3635        /*
3636         * Does target exist?
3637         */
3638        if (tzp) {
3639                /*
3640                 * Source and target must be the same type.
3641                 */
3642                if (ZTOV(szp)->v_type == VDIR) {
3643                        if (ZTOV(tzp)->v_type != VDIR) {
3644                                error = SET_ERROR(ENOTDIR);
3645                                goto out;
3646                        }
3647                } else {
3648                        if (ZTOV(tzp)->v_type == VDIR) {
3649                                error = SET_ERROR(EISDIR);
3650                                goto out;
3651                        }
3652                }
3653                /*
3654                 * POSIX dictates that when the source and target
3655                 * entries refer to the same file object, rename
3656                 * must do nothing and exit without error.
3657                 */
3658                if (szp->z_id == tzp->z_id) {
3659                        error = 0;
3660                        goto out;
3661                }
3662        }
3663
3664        vnevent_pre_rename_src(ZTOV(szp), sdvp, snm, ct);
3665        if (tzp)
3666                vnevent_pre_rename_dest(ZTOV(tzp), tdvp, tnm, ct);
3667
3668        /*
3669         * notify the target directory if it is not the same
3670         * as source directory.
3671         */
```



```
3672        if (tdvp != sdvp) {
3673                vnevent_pre_rename_dest_dir(tdvp, ZTOV(szp), tnm, ct);
3674        }
3675
3676        tx = dmu_tx_create(zfsvfs->z_os);
3677        dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
3678        dmu_tx_hold_sa(tx, sdzp->z_sa_hdl, B_FALSE);
3679        dmu_tx_hold_zap(tx, sdzp->z_id, FALSE, snm);
3680        dmu_tx_hold_zap(tx, tdzp->z_id, TRUE, tnm);
3681        if (sdzp != tdzp) {
3682                dmu_tx_hold_sa(tx, tdzp->z_sa_hdl, B_FALSE);
3683                zfs_sa_upgrade_txholds(tx, tdzp);
3684        }
3685        if (tzp) {
3686                dmu_tx_hold_sa(tx, tzp->z_sa_hdl, B_FALSE);
3687                zfs_sa_upgrade_txholds(tx, tzp);
3688        }
3689
3690        zfs_sa_upgrade_txholds(tx, szp);
3691        dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
3692        error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
3693        if (error) {
3694                if (zl != NULL)
3695                        zfs_rename_unlock(&zl);
3696                zfs_dirent_unlock(sdl);
3697                zfs_dirent_unlock(tdl);
3698
3699                if (sdzp == tdzp)
3700                        rw_exit(&sdzp->z_name_lock);
3701
3702                VN_RELE(ZTOV(szp));
3703                if (tzp)
3704                        VN_RELE(ZTOV(tzp));
3705                if (error == ERESTART) {
3706                        waited = B_TRUE;
3707                        dmu_tx_wait(tx);
3708                        dmu_tx_abort(tx);
3709                        goto top;
3710                }
3711                dmu_tx_abort(tx);
3712                ZFS_EXIT(zfsvfs);
3713                return (error);
3714        }
3715
3716        if (tzp)        /* Attempt to remove the existing target */
3717                error = rm_err = zfs_link_destroy(tdl, tzp, tx, zflg, NULL);
3718
3719        if (error == 0) {
3720                error = zfs_link_create(tdl, szp, tx, ZRENAMING);
3721                if (error == 0) {
3722                        szp->z_pflags |= ZFS_AV_MODIFIED;
3723
3724                        error = sa_update(szp->z_sa_hdl, SA_ZPL_FLAGS(zfsvfs),
3725                            (void *)&szp->z_pflags, sizeof (uint64_t), tx);
3726                        ASSERT0(error);
3727
3728                        error = zfs_link_destroy(sdl, szp, tx, ZRENAMING, NULL);
3729                        if (error == 0) {
3730                                zfs_log_rename(zilog, tx, TX_RENAME |
3731                                    (flags & FIGNORECASE ? TX_CI : 0), sdzp,
3732                                    sdl->dl_name, tdzp, tdl->dl_name, szp);
3733
3734                                /*
3735                                 * Update path information for the target vnode
3736                                 */
3737                                vn_renamepath(tdvp, ZTOV(szp), tnm,
```

```
3738                                    strlen(tnm));
3739                        } else {
3740                                /*
3741                                 * At this point, we have successfully created
3742                                 * the target name, but have failed to remove
3743                                 * the source name.  Since the create was done
3744                                 * with the ZRENAMING flag, there are
3745                                 * complications; for one, the link count is
3746                                 * wrong.  The easiest way to deal with this
3747                                 * is to remove the newly created target, and
3748                                 * return the original error.  This must
3749                                 * succeed; fortunately, it is very unlikely to
3750                                 * fail, since we just created it.
3751                                 */
3752                                VERIFY3U(zfs_link_destroy(tdl, szp, tx,
3753                                    ZRENAMING, NULL), ==, 0);
3754                        }
3755                }
3756        }

3758        dmu_tx_commit(tx);

3760        if (tzp && rm_err == 0)
3761                vnevent_rename_dest(ZTOV(tzp), tdvp, tnm, ct);

3763        if (error == 0) {
3764                vnevent_rename_src(ZTOV(szp), sdvp, snm, ct);
3765                /* notify the target dir if it is not the same as source dir */
3766                if (tdvp != sdvp)
3767                        vnevent_rename_dest_dir(tdvp, ct);
3768        }
3769 out:
3770        if (zl != NULL)
3771                zfs_rename_unlock(&zl);

3773        zfs_dirent_unlock(sdl);
3774        zfs_dirent_unlock(tdl);

3776        if (sdzp == tdzp)
3777                rw_exit(&sdzp->z_name_lock);


3780        VN_RELE(ZTOV(szp));
3781        if (tzp)
3782                VN_RELE(ZTOV(tzp));

3784        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3785                zil_commit(zilog, 0);

3787        ZFS_EXIT(zfsvfs);
3788        return (error);
3789 }

3791 /*
3792  * Insert the indicated symbolic reference entry into the directory.
3793  *
3794  *      IN:     dvp     - Directory to contain new symbolic link.
3795  *              link    - Name for new symlink entry.
3796  *              vap     - Attributes of new entry.
3797  *              cr      - credentials of caller.
3798  *              ct      - caller context
3799  *              flags   - case flags
3800  *
3801  *      RETURN: 0 on success, error code on failure.
3802  *
3803  * Timestamps:
```



```
3804  *      dvp - ctime|mtime updated
3805  */
3806 /*ARGSUSED*/
3807 static int
3808 zfs_symlink(vnode_t *dvp, char *name, vattr_t *vap, char *link, cred_t *cr,
3809     caller_context_t *ct, int flags)
3810 {
3811        znode_t         *zp, *dzp = VTOZ(dvp);
3812        zfs_dirlock_t   *dl;
3813        dmu_tx_t        *tx;
3814        zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
3815        zilog_t         *zilog;
3816        uint64_t        len = strlen(link);
3817        int             error;
3818        int             zflg = ZNEW;
3819        zfs_acl_ids_t   acl_ids;
3820        boolean_t       fuid_dirtied;
3821        uint64_t        txtype = TX_SYMLINK;
3822        boolean_t       waited = B_FALSE;

3824        ASSERT(vap->va_type == VLNK);

3826        ZFS_ENTER(zfsvfs);
3827        ZFS_VERIFY_ZP(dzp);
3828        zilog = zfsvfs->z_log;

3830        if (zfsvfs->z_utf8 && u8_validate(name, strlen(name),
3831            NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3832                ZFS_EXIT(zfsvfs);
3833                return (SET_ERROR(EILSEQ));
3834        }
3835        if (flags & FIGNORECASE)
3836                zflg |= ZCILOOK;

3838        if (len > MAXPATHLEN) {
3839                ZFS_EXIT(zfsvfs);
3840                return (SET_ERROR(ENAMETOOLONG));
3841        }

3843        if ((error = zfs_acl_ids_create(dzp, 0,
3844            vap, cr, NULL, &acl_ids)) != 0) {
3845                ZFS_EXIT(zfsvfs);
3846                return (error);
3847        }
3848 top:
3849        /*
3850         * Attempt to lock directory; fail if entry already exists.
3851         */
3852        error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg, NULL, NULL);
3853        if (error) {
3854                zfs_acl_ids_free(&acl_ids);
3855                ZFS_EXIT(zfsvfs);
3856                return (error);
3857        }

3859        if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
3860                zfs_acl_ids_free(&acl_ids);
3861                zfs_dirent_unlock(dl);
3862                ZFS_EXIT(zfsvfs);
3863                return (error);
3864        }

3866        if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
3867                zfs_acl_ids_free(&acl_ids);
3868                zfs_dirent_unlock(dl);
3869                ZFS_EXIT(zfsvfs);
```

```
3870                return (SET_ERROR(EDQUOT));
3871        }
3872        tx = dmu_tx_create(zfsvfs->z_os);
3873        fuid_dirtied = zfsvfs->z_fuid_dirty;
3874        dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0, MAX(1, len));
3875        dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
3876        dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
3877            ZFS_SA_BASE_ATTR_SIZE + len);
3878        dmu_tx_hold_sa(tx, dzp->z_sa_hdl, B_FALSE);
3879        if (!zfsvfs->z_use_sa && acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3880                dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
3881                    acl_ids.z_aclp->z_acl_bytes);
3882        }
3883        if (fuid_dirtied)
3884                zfs_fuid_txhold(zfsvfs, tx);
3885        error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
3886        if (error) {
3887                zfs_dirent_unlock(dl);
3888                if (error == ERESTART) {
3889                        waited = B_TRUE;
3890                        dmu_tx_wait(tx);
3891                        dmu_tx_abort(tx);
3892                        goto top;
3893                }
3894                zfs_acl_ids_free(&acl_ids);
3895                dmu_tx_abort(tx);
3896                ZFS_EXIT(zfsvfs);
3897                return (error);
3898        }

3900        /*
3901         * Create a new object for the symlink.
3902         * for version 4 ZPL datsets the symlink will be an SA attribute
3903         */
3904        zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

3906        if (fuid_dirtied)
3907                zfs_fuid_sync(zfsvfs, tx);

3909        mutex_enter(&zp->z_lock);
3910        if (zp->z_is_sa)
3911                error = sa_update(zp->z_sa_hdl, SA_ZPL_SYMLINK(zfsvfs),
3912                    link, len, tx);
3913        else
3914                zfs_sa_symlink(zp, link, len, tx);
3915        mutex_exit(&zp->z_lock);

3917        zp->z_size = len;
3918        (void) sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zfsvfs),
3919            &zp->z_size, sizeof (zp->z_size), tx);
3920        /*
3921         * Insert the new object into the directory.
3922         */
3923        (void) zfs_link_create(dl, zp, tx, ZNEW);

3925        if (flags & FIGNORECASE)
3926                txtype |= TX_CI;
3927        zfs_log_symlink(zilog, tx, txtype, dzp, zp, name, link);

3929        zfs_acl_ids_free(&acl_ids);

3931        dmu_tx_commit(tx);

3933        zfs_dirent_unlock(dl);

3935        VN_RELE(ZTOV(zp));
```



```
3937        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3938                zil_commit(zilog, 0);

3940        ZFS_EXIT(zfsvfs);
3941        return (error);
3942 }

3944 /*
3945  * Return, in the buffer contained in the provided uio structure,
3946  * the symbolic path referred to by vp.
3947  *
3948  *      IN:     vp      - vnode of symbolic link.
3949  *              uio     - structure to contain the link path.
3950  *              cr      - credentials of caller.
3951  *              ct      - caller context
3952  *
3953  *      OUT:    uio     - structure containing the link path.
3954  *
3955  *      RETURN: 0 on success, error code on failure.
3956  *
3957  * Timestamps:
3958  *      vp - atime updated
3959  */
3960 /* ARGSUSED */
3961 static int
3962 zfs_readlink(vnode_t *vp, uio_t *uio, cred_t *cr, caller_context_t *ct)
3963 {
3964        znode_t         *zp = VTOZ(vp);
3965        zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
3966        int             error;

3968        ZFS_ENTER(zfsvfs);
3969        ZFS_VERIFY_ZP(zp);

3971        mutex_enter(&zp->z_lock);
3972        if (zp->z_is_sa)
3973                error = sa_lookup_uio(zp->z_sa_hdl,
3974                    SA_ZPL_SYMLINK(zfsvfs), uio);
3975        else
3976                error = zfs_sa_readlink(zp, uio);
3977        mutex_exit(&zp->z_lock);

3979        ZFS_ACCESSTIME_STAMP(zfsvfs, zp);

3981        ZFS_EXIT(zfsvfs);
3982        return (error);
3983 }

3985 /*
3986  * Insert a new entry into directory tdvp referencing svp.
3987  *
3988  *      IN:     tdvp    - Directory to contain new entry.
3989  *              svp     - vnode of new entry.
3990  *              name    - name of new entry.
3991  *              cr      - credentials of caller.
3992  *              ct      - caller context
3993  *
3994  *      RETURN: 0 on success, error code on failure.
3995  *
3996  * Timestamps:
3997  *      tdvp - ctime|mtime updated
3998  *       svp - ctime updated
3999  */
4000 /* ARGSUSED */
4001 static int
```

```
4002 zfs_link(vnode_t *tdvp, vnode_t *svp, char *name, cred_t *cr,
4003     caller_context_t *ct, int flags)
4004 {
4005         znode_t         *dzp = VTOZ(tdvp);
4006         znode_t         *tzp, *szp;
4007         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
4008         zilog_t         *zilog;
4009         zfs_dirlock_t   *dl;
4010         dmu_tx_t        *tx;
4011         vnode_t         *realvp;
4012         int             error;
4013         int             zf = ZNEW;
4014         uint64_t        parent;
4015         uid_t           owner;
4016         boolean_t       waited = B_FALSE;

4018         ASSERT(tdvp->v_type == VDIR);

4020         ZFS_ENTER(zfsvfs);
4021         ZFS_VERIFY_ZP(dzp);
4022         zilog = zfsvfs->z_log;

4024         if (VOP_REALVP(svp, &realvp, ct) == 0)
4025                 svp = realvp;

4027         /*
4028          * POSIX dictates that we return EPERM here.
4029          * Better choices include ENOTSUP or EISDIR.
4030          */
4031         if (svp->v_type == VDIR) {
4032                 ZFS_EXIT(zfsvfs);
4033                 return (SET_ERROR(EPERM));
4034         }

4036         szp = VTOZ(svp);
4037         ZFS_VERIFY_ZP(szp);

4039         /*
4040          * We check z_zfsvfs rather than v_vfsp here, because snapshots and the
4041          * ctldir appear to have the same v_vfsp.
4042          */
4043         if (szp->z_zfsvfs != zfsvfs || zfsctl_is_node(svp)) {
4044                 ZFS_EXIT(zfsvfs);
4045                 return (SET_ERROR(EXDEV));
4046         }

4048         /* Prevent links to .zfs/shares files */

4050         if ((error = sa_lookup(szp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
4051             &parent, sizeof (uint64_t))) != 0) {
4052                 ZFS_EXIT(zfsvfs);
4053                 return (error);
4054         }
4055         if (parent == zfsvfs->z_shares_dir) {
4056                 ZFS_EXIT(zfsvfs);
4057                 return (SET_ERROR(EPERM));
4058         }

4060         if (zfsvfs->z_utf8 && u8_validate(name,
4061             strlen(name), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
4062                 ZFS_EXIT(zfsvfs);
4063                 return (SET_ERROR(EILSEQ));
4064         }
4065         if (flags & FIGNORECASE)
4066                 zf |= ZCILOOK;
```



```
4068         /*
4069          * We do not support links between attributes and non-attributes
4070          * because of the potential security risk of creating links
4071          * into "normal" file space in order to circumvent restrictions
4072          * imposed in attribute space.
4073          */
4074         if ((szp->z_pflags & ZFS_XATTR) != (dzp->z_pflags & ZFS_XATTR)) {
4075                 ZFS_EXIT(zfsvfs);
4076                 return (SET_ERROR(EINVAL));
4077         }


4080         owner = zfs_fuid_map_id(zfsvfs, szp->z_uid, cr, ZFS_OWNER);
4081         if (owner != crgetuid(cr) && secpolicy_basic_link(cr) != 0) {
4082                 ZFS_EXIT(zfsvfs);
4083                 return (SET_ERROR(EPERM));
4084         }

4086         if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
4087                 ZFS_EXIT(zfsvfs);
4088                 return (error);
4089         }

4091 top:
4092         /*
4093          * Attempt to lock directory; fail if entry already exists.
4094          */
4095         error = zfs_dirent_lock(&dl, dzp, name, &tzp, zf, NULL, NULL);
4096         if (error) {
4097                 ZFS_EXIT(zfsvfs);
4098                 return (error);
4099         }

4101         tx = dmu_tx_create(zfsvfs->z_os);
4102         dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
4103         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
4104         zfs_sa_upgrade_txholds(tx, szp);
4105         zfs_sa_upgrade_txholds(tx, dzp);
4106         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
4107         if (error) {
4108                 zfs_dirent_unlock(dl);
4109                 if (error == ERESTART) {
4110                         waited = B_TRUE;
4111                         dmu_tx_wait(tx);
4112                         dmu_tx_abort(tx);
4113                         goto top;
4114                 }
4115                 dmu_tx_abort(tx);
4116                 ZFS_EXIT(zfsvfs);
4117                 return (error);
4118         }

4120         error = zfs_link_create(dl, szp, tx, 0);

4122         if (error == 0) {
4123                 uint64_t txtype = TX_LINK;
4124                 if (flags & FIGNORECASE)
4125                         txtype |= TX_CI;
4126                 zfs_log_link(zilog, tx, txtype, dzp, szp, name);
4127         }

4129         dmu_tx_commit(tx);

4131         zfs_dirent_unlock(dl);

4133         if (error == 0) {
```

```
4134                 vnevent_link(svp, ct);
4135         }

4137         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
4138                 zil_commit(zilog, 0);

4140         ZFS_EXIT(zfsvfs);
4141         return (error);
4142 }

4144 /*
4145  * zfs_null_putapage() is used when the file system has been force
4146  * unmounted. It just drops the pages.
4147  */
4148 /* ARGSUSED */
4149 static int
4150 zfs_null_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp,
4151     size_t *lenp, int flags, cred_t *cr)
4152 {
4153         pvn_write_done(pp, B_INVAL|B_FORCE|B_ERROR);
4154         return (0);
4155 }

4157 /*
4158  * Push a page out to disk, klustering if possible.
4159  *
4160  *      IN:     vp      - file to push page to.
4161  *              pp      - page to push.
4162  *              flags   - additional flags.
4163  *              cr      - credentials of caller.
4164  *
4165  *      OUT:    offp    - start of range pushed.
4166  *              lenp    - len of range pushed.
4167  *
4168  *      RETURN: 0 on success, error code on failure.
4169  *
4170  * NOTE: callers must have locked the page to be pushed.  On
4171  * exit, the page (and all other pages in the kluster) must be
4172  * unlocked.
4173  */
4174 /* ARGSUSED */
4175 static int
4176 zfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp,
4177     size_t *lenp, int flags, cred_t *cr)
4178 {
4179         znode_t         *zp = VTOZ(vp);
4180         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4181         dmu_tx_t        *tx;
4182         u_offset_t      off, koff;
4183         size_t          len, klen;
4184         int             err;

4186         off = pp->p_offset;
4187         len = PAGESIZE;
4188         /*
4189          * If our blocksize is bigger than the page size, try to kluster
4190          * multiple pages so that we write a full block (thus avoiding
4191          * a read-modify-write).
4192          */
4193         if (off < zp->z_size && zp->z_blksz > PAGESIZE) {
4194                 klen = P2ROUNDUP((ulong_t)zp->z_blksz, PAGESIZE);
4195                 koff = ISP2(klen) ? P2ALIGN(off, (u_offset_t)klen) : 0;
4196                 ASSERT(koff <= zp->z_size);
4197                 if (koff + klen > zp->z_size)
4198                         klen = P2ROUNDUP(zp->z_size - koff, (uint64_t)PAGESIZE);
4199                 pp = pvn_write_kluster(vp, pp, &off, &len, koff, klen, flags);
```



```
4200         }
4201         ASSERT3U(btop(len), ==, btopr(len));

4203         /*
4204          * Can't push pages past end-of-file.
4205          */
4206         if (off >= zp->z_size) {
4207                 /* ignore all pages */
4208                 err = 0;
4209                 goto out;
4210         } else if (off + len > zp->z_size) {
4211                 int npages = btopr(zp->z_size - off);
4212                 page_t *trunc;

4214                 page_list_break(&pp, &trunc, npages);
4215                 /* ignore pages past end of file */
4216                 if (trunc)
4217                         pvn_write_done(trunc, flags);
4218                 len = zp->z_size - off;
4219         }

4221         if (zfs_owner_overquota(zfsvfs, zp, B_FALSE) ||
4222             zfs_owner_overquota(zfsvfs, zp, B_TRUE)) {
4223                 err = SET_ERROR(EDQUOT);
4224                 goto out;
4225         }
4226         tx = dmu_tx_create(zfsvfs->z_os);
4227         dmu_tx_hold_write(tx, zp->z_id, off, len);

4229         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
4230         zfs_sa_upgrade_txholds(tx, zp);
4231         err = dmu_tx_assign(tx, TXG_WAIT);
4232         if (err != 0) {
4233                 dmu_tx_abort(tx);
4234                 goto out;
4235         }

4237         if (zp->z_blksz <= PAGESIZE) {
4238                 caddr_t va = zfs_map_page(pp, S_READ);
4239                 ASSERT3U(len, <=, PAGESIZE);
4240                 dmu_write(zfsvfs->z_os, zp->z_id, off, len, va, tx);
4241                 zfs_unmap_page(pp, va);
4242         } else {
4243                 err = dmu_write_pages(zfsvfs->z_os, zp->z_id, off, len, pp, tx);
4244         }

4246         if (err == 0) {
4247                 uint64_t mtime[2], ctime[2];
4248                 sa_bulk_attr_t bulk[3];
4249                 int count = 0;

4251                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
4252                     &mtime, 16);
4253                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
4254                     &ctime, 16);
4255                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
4256                     &zp->z_pflags, 8);
4257                 zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
4258                     B_TRUE);
4259                 zfs_log_write(zfsvfs->z_log, tx, TX_WRITE, zp, off, len, 0);
4260         }
4261         dmu_tx_commit(tx);

4263 out:
4264         pvn_write_done(pp, (err ? B_ERROR : 0) | flags);
4265         if (offp)
```

```
4266                *offp = off;
4267        if (lenp)
4268                *lenp = len;

4270        return (err);
4271 }

4273 /*
4274  * Copy the portion of the file indicated from pages into the file.
4275  * The pages are stored in a page list attached to the files vnode.
4276  *
4277  *      IN:     vp      - vnode of file to push page data to.
4278  *              off     - position in file to put data.
4279  *              len     - amount of data to write.
4280  *              flags   - flags to control the operation.
4281  *              cr      - credentials of caller.
4282  *              ct      - caller context.
4283  *
4284  *      RETURN: 0 on success, error code on failure.
4285  *
4286  * Timestamps:
4287  *      vp - ctime|mtime updated
4288  */
4289 /*ARGSUSED*/
4290 static int
4291 zfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags, cred_t *cr,
4292     caller_context_t *ct)
4293 {
4294        znode_t         *zp = VTOZ(vp);
4295        zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4296        page_t          *pp;
4297        size_t          io_len;
4298        u_offset_t      io_off;
4299        uint_t          blksz;
4300        rl_t            *rl;
4301        int             error = 0;

4303        ZFS_ENTER(zfsvfs);
4304        ZFS_VERIFY_ZP(zp);

4306        /*
4307         * There's nothing to do if no data is cached.
4308         */
4309        if (!vn_has_cached_data(vp)) {
4310                ZFS_EXIT(zfsvfs);
4311                return (0);
4312        }

4314        /*
4315         * Align this request to the file block size in case we kluster.
4316         * XXX - this can result in pretty aggresive locking, which can
4317         * impact simultanious read/write access.  One option might be
4318         * to break up long requests (len == 0) into block-by-block
4319         * operations to get narrower locking.
4320         */
4321        blksz = zp->z_blksz;
4322        if (ISP2(blksz))
4323                io_off = P2ALIGN_TYPED(off, blksz, u_offset_t);
4324        else
4325                io_off = 0;
4326        if (len > 0 && ISP2(blksz))
4327                io_len = P2ROUNDUP_TYPED(len + (off - io_off), blksz, size_t);
4328        else
4329                io_len = 0;

4331        if (io_len == 0) {
```



```
4332                /*
4333                 * Search the entire vp list for pages >= io_off.
4334                 */
4335                rl = zfs_range_lock(zp, io_off, UINT64_MAX, RL_WRITER);
4336                error = pvn_vplist_dirty(vp, io_off, zfs_putapage, flags, cr);
4337                goto out;
4338        }
4339        rl = zfs_range_lock(zp, io_off, io_len, RL_WRITER);

4341        if (off > zp->z_size) {
4342                /* past end of file */
4343                zfs_range_unlock(rl);
4344                ZFS_EXIT(zfsvfs);
4345                return (0);
4346        }

4348        len = MIN(io_len, P2ROUNDUP(zp->z_size, PAGESIZE) - io_off);

4350        for (off = io_off; io_off < off + len; io_off += io_len) {
4351                if ((flags & B_INVAL) || ((flags & B_ASYNC) == 0)) {
4352                        pp = page_lookup(vp, io_off,
4353                            (flags & (B_INVAL | B_FREE)) ? SE_EXCL : SE_SHARED);
4354                } else {
4355                        pp = page_lookup_nowait(vp, io_off,
4356                            (flags & B_FREE) ? SE_EXCL : SE_SHARED);
4357                }

4359                if (pp != NULL && pvn_getdirty(pp, flags)) {
4360                        int err;

4362                        /*
4363                         * Found a dirty page to push
4364                         */
4365                        err = zfs_putapage(vp, pp, &io_off, &io_len, flags, cr);
4366                        if (err)
4367                                error = err;
4368                } else {
4369                        io_len = PAGESIZE;
4370                }
4371        }
4372 out:
4373        zfs_range_unlock(rl);
4374        if ((flags & B_ASYNC) == 0 || zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
4375                zil_commit(zfsvfs->z_log, zp->z_id);
4376        ZFS_EXIT(zfsvfs);
4377        return (error);
4378 }

4380 /*ARGSUSED*/
4381 void
4382 zfs_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
4383 {
4384        znode_t *zp = VTOZ(vp);
4385        zfsvfs_t *zfsvfs = zp->z_zfsvfs;
4386        int error;

4388        rw_enter(&zfsvfs->z_teardown_inactive_lock, RW_READER);
4389        if (zp->z_sa_hdl == NULL) {
4390                /*
4391                 * The fs has been unmounted, or we did a
4392                 * suspend/resume and this file no longer exists.
4393                 */
4394                if (vn_has_cached_data(vp)) {
4395                        (void) pvn_vplist_dirty(vp, 0, zfs_null_putapage,
4396                            B_INVAL, cr);
4397                }
```

```
4399                mutex_enter(&zp->z_lock);
4400                mutex_enter(&vp->v_lock);
4401                ASSERT(vp->v_count == 1);
4402                vp->v_count = 0;
4403                mutex_exit(&vp->v_lock);
4404                mutex_exit(&zp->z_lock);
4405                rw_exit(&zfsvfs->z_teardown_inactive_lock);
4406                zfs_znode_free(zp);
4407                return;
4408        }

4410        /*
4411         * Attempt to push any data in the page cache.  If this fails
4412         * we will get kicked out later in zfs_zinactive().
4413         */
4414        if (vn_has_cached_data(vp)) {
4415                (void) pvn_vplist_dirty(vp, 0, zfs_putapage, B_INVAL|B_ASYNC,
4416                    cr);
4417        }

4419        if (zp->z_atime_dirty && zp->z_unlinked == 0) {
4420                dmu_tx_t *tx = dmu_tx_create(zfsvfs->z_os);

4422                dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
4423                zfs_sa_upgrade_txholds(tx, zp);
4424                error = dmu_tx_assign(tx, TXG_WAIT);
4425                if (error) {
4426                        dmu_tx_abort(tx);
4427                } else {
4428                        mutex_enter(&zp->z_lock);
4429                        (void) sa_update(zp->z_sa_hdl, SA_ZPL_ATIME(zfsvfs),
4430                            (void *)&zp->z_atime, sizeof (zp->z_atime), tx);
4431                        zp->z_atime_dirty = 0;
4432                        mutex_exit(&zp->z_lock);
4433                        dmu_tx_commit(tx);
4434                }
4435        }

4437        zfs_zinactive(zp);
4438        rw_exit(&zfsvfs->z_teardown_inactive_lock);
4439 }

4441 /*
4442  * Bounds-check the seek operation.
4443  *
4444  *      IN:     vp      - vnode seeking within
4445  *              ooff    - old file offset
4446  *              noffp   - pointer to new file offset
4447  *              ct      - caller context
4448  *
4449  *      RETURN: 0 on success, EINVAL if new offset invalid.
4450  */
4451 /* ARGSUSED */
4452 static int
4453 zfs_seek(vnode_t *vp, offset_t ooff, offset_t *noffp,
4454     caller_context_t *ct)
4455 {
4456        if (vp->v_type == VDIR)
4457                return (0);
4458        return ((*noffp < 0 || *noffp > MAXOFFSET_T) ? EINVAL : 0);
4459 }

4461 /*
4462  * Pre-filter the generic locking function to trap attempts to place
4463  * a mandatory lock on a memory mapped file.
```



```
4464  */
4465 static int
4466 zfs_frlock(vnode_t *vp, int cmd, flock64_t *bfp, int flag, offset_t offset,
4467     flk_callback_t *flk_cbp, cred_t *cr, caller_context_t *ct)
4468 {
4469        znode_t *zp = VTOZ(vp);
4470        zfsvfs_t *zfsvfs = zp->z_zfsvfs;

4472        ZFS_ENTER(zfsvfs);
4473        ZFS_VERIFY_ZP(zp);

4475        /*
4476         * We are following the UFS semantics with respect to mapcnt
4477         * here: If we see that the file is mapped already, then we will
4478         * return an error, but we don't worry about races between this
4479         * function and zfs_map().
4480         */
4481        if (zp->z_mapcnt > 0 && MANDMODE(zp->z_mode)) {
4482                ZFS_EXIT(zfsvfs);
4483                return (SET_ERROR(EAGAIN));
4484        }
4485        ZFS_EXIT(zfsvfs);
4486        return (fs_frlock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
4487 }

4489 /*
4490  * If we can't find a page in the cache, we will create a new page
4491  * and fill it with file data.  For efficiency, we may try to fill
4492  * multiple pages at once (klustering) to fill up the supplied page
4493  * list.  Note that the pages to be filled are held with an exclusive
4494  * lock to prevent access by other threads while they are being filled.
4495  */
4496 static int
4497 zfs_fillpage(vnode_t *vp, u_offset_t off, struct seg *seg,
4498     caddr_t addr, page_t *pl[], size_t plsz, enum seg_rw rw)
4499 {
4500        znode_t *zp = VTOZ(vp);
4501        page_t *pp, *cur_pp;
4502        objset_t *os = zp->z_zfsvfs->z_os;
4503        u_offset_t io_off, total;
4504        size_t io_len;
4505        int err;

4507        if (plsz == PAGESIZE || zp->z_blksz <= PAGESIZE) {
4508                /*
4509                 * We only have a single page, don't bother klustering
4510                 */
4511                io_off = off;
4512                io_len = PAGESIZE;
4513                pp = page_create_va(vp, io_off, io_len,
4514                    PG_EXCL | PG_WAIT, seg, addr);
4515        } else {
4516                /*
4517                 * Try to find enough pages to fill the page list
4518                 */
4519                pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
4520                    &io_len, off, plsz, 0);
4521        }
4522        if (pp == NULL) {
4523                /*
4524                 * The page already exists, nothing to do here.
4525                 */
4526                *pl = NULL;
4527                return (0);
4528        }
```

```
4530         /*
4531          * Fill the pages in the kluster.
4532          */
4533         cur_pp = pp;
4534         for (total = io_off + io_len; io_off < total; io_off += PAGESIZE) {
4535                 caddr_t va;

4537                 ASSERT3U(io_off, ==, cur_pp->p_offset);
4538                 va = zfs_map_page(cur_pp, S_WRITE);
4539                 err = dmu_read(os, zp->z_id, io_off, PAGESIZE, va,
4540                     DMU_READ_PREFETCH);
4541                 zfs_unmap_page(cur_pp, va);
4542                 if (err) {
4543                         /* On error, toss the entire kluster */
4544                         pvn_read_done(pp, B_ERROR);
4545                         /* convert checksum errors into IO errors */
4546                         if (err == ECKSUM)
4547                                 err = SET_ERROR(EIO);
4548                         return (err);
4549                 }
4550                 cur_pp = cur_pp->p_next;
4551         }

4553         /*
4554          * Fill in the page list array from the kluster starting
4555          * from the desired offset `off'.
4556          * NOTE: the page list will always be null terminated.
4557          */
4558         pvn_plist_init(pp, pl, plsz, off, io_len, rw);
4559         ASSERT(pl == NULL || (*pl)->p_offset == off);

4561         return (0);
4562 }

4564 /*
4565  * Return pointers to the pages for the file region [off, off + len]
4566  * in the pl array.  If plsz is greater than len, this function may
4567  * also return page pointers from after the specified region
4568  * (i.e. the region [off, off + plsz]).  These additional pages are
4569  * only returned if they are already in the cache, or were created as
4570  * part of a klustered read.
4571  *
4572  *      IN:     vp      - vnode of file to get data from.
4573  *              off     - position in file to get data from.
4574  *              len     - amount of data to retrieve.
4575  *              plsz    - length of provided page list.
4576  *              seg     - segment to obtain pages for.
4577  *              addr    - virtual address of fault.
4578  *              rw      - mode of created pages.
4579  *              cr      - credentials of caller.
4580  *              ct      - caller context.
4581  *
4582  *      OUT:    protp   - protection mode of created pages.
4583  *              pl      - list of pages created.
4584  *
4585  *      RETURN: 0 on success, error code on failure.
4586  *
4587  * Timestamps:
4588  *      vp - atime updated
4589  */
4590 /* ARGSUSED */
4591 static int
4592 zfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
4593     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
4594     enum seg_rw rw, cred_t *cr, caller_context_t *ct)
4595 {
```



```
4596         znode_t         *zp = VTOZ(vp);
4597         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4598         page_t          **pl0 = pl;
4599         int             err = 0;

4601         /* we do our own caching, faultahead is unnecessary */
4602         if (pl == NULL)
4603                 return (0);
4604         else if (len > plsz)
4605                 len = plsz;
4606         else
4607                 len = P2ROUNDUP(len, PAGESIZE);
4608         ASSERT(plsz >= len);

4610         ZFS_ENTER(zfsvfs);
4611         ZFS_VERIFY_ZP(zp);

4613         if (protp)
4614                 *protp = PROT_ALL;

4616         /*
4617          * Loop through the requested range [off, off + len) looking
4618          * for pages.  If we don't find a page, we will need to create
4619          * a new page and fill it with data from the file.
4620          */
4621         while (len > 0) {
4622                 if (*pl = page_lookup(vp, off, SE_SHARED))
4623                         *(pl+1) = NULL;
4624                 else if (err = zfs_fillpage(vp, off, seg, addr, pl, plsz, rw))
4625                         goto out;
4626                 while (*pl) {
4627                         ASSERT3U((*pl)->p_offset, ==, off);
4628                         off += PAGESIZE;
4629                         addr += PAGESIZE;
4630                         if (len > 0) {
4631                                 ASSERT3U(len, >=, PAGESIZE);
4632                                 len -= PAGESIZE;
4633                         }
4634                         ASSERT3U(plsz, >=, PAGESIZE);
4635                         plsz -= PAGESIZE;
4636                         pl++;
4637                 }
4638         }

4640         /*
4641          * Fill out the page array with any pages already in the cache.
4642          */
4643         while (plsz > 0 &&
4644             (*pl++ = page_lookup_nowait(vp, off, SE_SHARED))) {
4645                         off += PAGESIZE;
4646                         plsz -= PAGESIZE;
4647         }
4648 out:
4649         if (err) {
4650                 /*
4651                  * Release any pages we have previously locked.
4652                  */
4653                 while (pl > pl0)
4654                         page_unlock(*--pl);
4655         } else {
4656                 ZFS_ACCESSTIME_STAMP(zfsvfs, zp);
4657         }

4659         *pl = NULL;

4661         ZFS_EXIT(zfsvfs);
```

```
4662         return (err);
4663 }
4664
4665 /*
4666  * Request a memory map for a section of a file.  This code interacts
4667  * with common code and the VM system as follows:
4668  *
4669  * - common code calls mmap(), which ends up in smmap_common()
4670  * - this calls VOP_MAP(), which takes you into (say) zfs
4671  * - zfs_map() calls as_map(), passing segvn_create() as the callback
4672  * - segvn_create() creates the new segment and calls VOP_ADDMAP()
4673  * - zfs_addmap() updates z_mapcnt
4674  */
4675 /*ARGSUSED*/
4676 static int
4677 zfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
4678     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
4679     caller_context_t *ct)
4680 {
4681         znode_t *zp = VTOZ(vp);
4682         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
4683         segvn_crargs_t  vn_a;
4684         int             error;
4685
4686         ZFS_ENTER(zfsvfs);
4687         ZFS_VERIFY_ZP(zp);
4688
4689         if ((prot & PROT_WRITE) && (zp->z_pflags &
4690             (ZFS_IMMUTABLE | ZFS_READONLY | ZFS_APPENDONLY))) {
4691                 ZFS_EXIT(zfsvfs);
4692                 return (SET_ERROR(EPERM));
4693         }
4694
4695         if ((prot & (PROT_READ | PROT_EXEC)) &&
4696             (zp->z_pflags & ZFS_AV_QUARANTINED)) {
4697                 ZFS_EXIT(zfsvfs);
4698                 return (SET_ERROR(EACCES));
4699         }
4700
4701         if (vp->v_flag & VNOMAP) {
4702                 ZFS_EXIT(zfsvfs);
4703                 return (SET_ERROR(ENOSYS));
4704         }
4705
4706         if (off < 0 || len > MAXOFFSET_T - off) {
4707                 ZFS_EXIT(zfsvfs);
4708                 return (SET_ERROR(ENXIO));
4709         }
4710
4711         if (vp->v_type != VREG) {
4712                 ZFS_EXIT(zfsvfs);
4713                 return (SET_ERROR(ENODEV));
4714         }
4715
4716         /*
4717          * If file is locked, disallow mapping.
4718          */
4719         if (MANDMODE(zp->z_mode) && vn_has_flocks(vp)) {
4720                 ZFS_EXIT(zfsvfs);
4721                 return (SET_ERROR(EAGAIN));
4722         }
4723
4724         as_rangelock(as);
4725         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
4726         if (error != 0) {
4727                 as_rangeunlock(as);
```



```
4728                 ZFS_EXIT(zfsvfs);
4729                 return (error);
4730         }
4731
4732         vn_a.vp = vp;
4733         vn_a.offset = (u_offset_t)off;
4734         vn_a.type = flags & MAP_TYPE;
4735         vn_a.prot = prot;
4736         vn_a.maxprot = maxprot;
4737         vn_a.cred = cr;
4738         vn_a.amp = NULL;
4739         vn_a.flags = flags & ~MAP_TYPE;
4740         vn_a.szc = 0;
4741         vn_a.lgrp_mem_policy_flags = 0;
4742
4743         error = as_map(as, *addrp, len, segvn_create, &vn_a);
4744
4745         as_rangeunlock(as);
4746         ZFS_EXIT(zfsvfs);
4747         return (error);
4748 }
4749
4750 /* ARGSUSED */
4751 static int
4752 zfs_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
4753     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
4754     caller_context_t *ct)
4755 {
4756         uint64_t pages = btopr(len);
4757
4758         atomic_add_64(&VTOZ(vp)->z_mapcnt, pages);
4759         return (0);
4760 }
4761
4762 /*
4763  * The reason we push dirty pages as part of zfs_delmap() is so that we get a
4764  * more accurate mtime for the associated file.  Since we don't have a way of
4765  * detecting when the data was actually modified, we have to resort to
4766  * heuristics.  If an explicit msync() is done, then we mark the mtime when the
4767  * last page is pushed.  The problem occurs when the msync() call is omitted,
4768  * which by far the most common case:
4769  *
4770  *      open()
4771  *      mmap()
4772  *      <modify memory>
4773  *      munmap()
4774  *      close()
4775  *      <time lapse>
4776  *      putpage() via fsflush
4777  *
4778  * If we wait until fsflush to come along, we can have a modification time that
4779  * is some arbitrary point in the future.  In order to prevent this in the
4780  * common case, we flush pages whenever a (MAP_SHARED, PROT_WRITE) mapping is
4781  * torn down.
4782  */
4783 /* ARGSUSED */
4784 static int
4785 zfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
4786     size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
4787     caller_context_t *ct)
4788 {
4789         uint64_t pages = btopr(len);
4790
4791         ASSERT3U(VTOZ(vp)->z_mapcnt, >=, pages);
4792         atomic_add_64(&VTOZ(vp)->z_mapcnt, -pages);
```

```
4794         if ((flags & MAP_SHARED) && (prot & PROT_WRITE) &&
4795             vn_has_cached_data(vp))
4796                 (void) VOP_PUTPAGE(vp, off, len, B_ASYNC, cr, ct);

4798         return (0);
4799 }

4801 /*
4802  * Free or allocate space in a file.  Currently, this function only
4803  * supports the `F_FREESP' command.  However, this command is somewhat
4804  * misnamed, as its functionality includes the ability to allocate as
4805  * well as free space.
4806  *
4807  *      IN:     vp      - vnode of file to free data in.
4808  *              cmd     - action to take (only F_FREESP supported).
4809  *              bfp     - section of file to free/alloc.
4810  *              flag    - current file open mode flags.
4811  *              offset  - current file offset.
4812  *              cr      - credentials of caller [UNUSED].
4813  *              ct      - caller context.
4814  *
4815  *      RETURN: 0 on success, error code on failure.
4816  *
4817  * Timestamps:
4818  *      vp - ctime|mtime updated
4819  */
4820 /* ARGSUSED */
4821 static int
4822 zfs_space(vnode_t *vp, int cmd, flock64_t *bfp, int flag,
4823     offset_t offset, cred_t *cr, caller_context_t *ct)
4824 {
4825         znode_t         *zp = VTOZ(vp);
4826         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4827         uint64_t        off, len;
4828         int             error;

4830         ZFS_ENTER(zfsvfs);
4831         ZFS_VERIFY_ZP(zp);

4833         if (cmd != F_FREESP) {
4834                 ZFS_EXIT(zfsvfs);
4835                 return (SET_ERROR(EINVAL));
4836         }

4838         /*
4839          * In a case vp->v_vfsp != zp->z_zfsvfs->z_vfs (e.g. snapshots) our
4840          * callers might not be able to detect properly that we are read-only,
4841          * so check it explicitly here.
4842          */
4843         if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
4844                 ZFS_EXIT(zfsvfs);
4845                 return (SET_ERROR(EROFS));
4846         }

4848         if (error = convoff(vp, bfp, 0, offset)) {
4849                 ZFS_EXIT(zfsvfs);
4850                 return (error);
4851         }

4853         if (bfp->l_len < 0) {
4854                 ZFS_EXIT(zfsvfs);
4855                 return (SET_ERROR(EINVAL));
4856         }

4858         off = bfp->l_start;
4859         len = bfp->l_len; /* 0 means from off to end of file */
```



```
4861         error = zfs_freesp(zp, off, len, flag, TRUE);

4863         if (error == 0 && off == 0 && len == 0)
4864                 vnevent_truncate(ZTOV(zp), ct);

4866         ZFS_EXIT(zfsvfs);
4867         return (error);
4868 }

4870 /*ARGSUSED*/
4871 static int
4872 zfs_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
4873 {
4874         znode_t         *zp = VTOZ(vp);
4875         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4876         uint32_t        gen;
4877         uint64_t        gen64;
4878         uint64_t        object = zp->z_id;
4879         zfid_short_t    *zfid;
4880         int             size, i, error;

4882         ZFS_ENTER(zfsvfs);
4883         ZFS_VERIFY_ZP(zp);

4885         if ((error = sa_lookup(zp->z_sa_hdl, SA_ZPL_GEN(zfsvfs),
4886             &gen64, sizeof (uint64_t))) != 0) {
4887                 ZFS_EXIT(zfsvfs);
4888                 return (error);
4889         }

4891         gen = (uint32_t)gen64;

4893         size = (zfsvfs->z_parent != zfsvfs) ? LONG_FID_LEN : SHORT_FID_LEN;
4894         if (fidp->fid_len < size) {
4895                 fidp->fid_len = size;
4896                 ZFS_EXIT(zfsvfs);
4897                 return (SET_ERROR(ENOSPC));
4898         }

4900         zfid = (zfid_short_t *)fidp;

4902         zfid->zf_len = size;

4904         for (i = 0; i < sizeof (zfid->zf_object); i++)
4905                 zfid->zf_object[i] = (uint8_t)(object >> (8 * i));

4907         /* Must have a non-zero generation number to distinguish from .zfs */
4908         if (gen == 0)
4909                 gen = 1;
4910         for (i = 0; i < sizeof (zfid->zf_gen); i++)
4911                 zfid->zf_gen[i] = (uint8_t)(gen >> (8 * i));

4913         if (size == LONG_FID_LEN) {
4914                 uint64_t        objsetid = dmu_objset_id(zfsvfs->z_os);
4915                 zfid_long_t     *zlfid;

4917                 zlfid = (zfid_long_t *)fidp;

4919                 for (i = 0; i < sizeof (zlfid->zf_setid); i++)
4920                         zlfid->zf_setid[i] = (uint8_t)(objsetid >> (8 * i));

4922                 /* XXX - this should be the generation number for the objset */
4923                 for (i = 0; i < sizeof (zlfid->zf_setgen); i++)
4924                         zlfid->zf_setgen[i] = 0;
4925         }
```

```
4927         ZFS_EXIT(zfsvfs);
4928         return (0);
4929 }

4931 static int
4932 zfs_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
4933     caller_context_t *ct)
4934 {
4935         znode_t         *zp, *xzp;
4936         zfsvfs_t        *zfsvfs;
4937         zfs_dirlock_t   *dl;
4938         int             error;

4940         switch (cmd) {
4941         case _PC_LINK_MAX:
4942                 *valp = ULONG_MAX;
4943                 return (0);

4945         case _PC_FILESIZEBITS:
4946                 *valp = 64;
4947                 return (0);

4949         case _PC_XATTR_EXISTS:
4950                 zp = VTOZ(vp);
4951                 zfsvfs = zp->z_zfsvfs;
4952                 ZFS_ENTER(zfsvfs);
4953                 ZFS_VERIFY_ZP(zp);
4954                 *valp = 0;
4955                 error = zfs_dirent_lock(&dl, zp, "", &xzp,
4956                     ZXATTR | ZEXISTS | ZSHARED, NULL, NULL);
4957                 if (error == 0) {
4958                         zfs_dirent_unlock(dl);
4959                         if (!zfs_dirempty(xzp))
4960                                 *valp = 1;
4961                         VN_RELE(ZTOV(xzp));
4962                 } else if (error == ENOENT) {
4963                         /*
4964                          * If there aren't extended attributes, it's the
4965                          * same as having zero of them.
4966                          */
4967                         error = 0;
4968                 }
4969                 ZFS_EXIT(zfsvfs);
4970                 return (error);

4972         case _PC_SATTR_ENABLED:
4973         case _PC_SATTR_EXISTS:
4974                 *valp = vfs_has_feature(vp->v_vfsp, VFSFT_SYSATTR_VIEWS) &&
4975                     (vp->v_type == VREG || vp->v_type == VDIR);
4976                 return (0);

4978         case _PC_ACCESS_FILTERING:
4979                 *valp = vfs_has_feature(vp->v_vfsp, VFSFT_ACCESS_FILTER) &&
4980                     vp->v_type == VDIR;
4981                 return (0);

4983         case _PC_ACL_ENABLED:
4984                 *valp = _ACL_ACE_ENABLED;
4985                 return (0);

4987         case _PC_MIN_HOLE_SIZE:
4988                 *valp = (ulong_t)SPA_MINBLOCKSIZE;
4989                 return (0);

4991         case _PC_TIMESTAMP_RESOLUTION:
```



```
4992                 /* nanosecond timestamp resolution */
4993                 *valp = 1L;
4994                 return (0);

4996         default:
4997                 return (fs_pathconf(vp, cmd, valp, cr, ct));
4998         }
4999 }

5001 /*ARGSUSED*/
5002 static int
5003 zfs_getsecattr(vnode_t *vp, vsecattr_t *vsecp, int flag, cred_t *cr,
5004     caller_context_t *ct)
5005 {
5006         znode_t *zp = VTOZ(vp);
5007         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
5008         int error;
5009         boolean_t skipaclchk = (flag & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;

5011         ZFS_ENTER(zfsvfs);
5012         ZFS_VERIFY_ZP(zp);
5013         error = zfs_getacl(zp, vsecp, skipaclchk, cr);
5014         ZFS_EXIT(zfsvfs);

5016         return (error);
5017 }

5019 /*ARGSUSED*/
5020 static int
5021 zfs_setsecattr(vnode_t *vp, vsecattr_t *vsecp, int flag, cred_t *cr,
5022     caller_context_t *ct)
5023 {
5024         znode_t *zp = VTOZ(vp);
5025         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
5026         int error;
5027         boolean_t skipaclchk = (flag & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
5028         zilog_t *zilog = zfsvfs->z_log;

5030         ZFS_ENTER(zfsvfs);
5031         ZFS_VERIFY_ZP(zp);

5033         error = zfs_setacl(zp, vsecp, skipaclchk, cr);

5035         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
5036                 zil_commit(zilog, 0);

5038         ZFS_EXIT(zfsvfs);
5039         return (error);
5040 }

5042 /*
5043  * The smallest read we may consider to loan out an arcbuf.
5044  * This must be a power of 2.
5045  */
5046 int zcr_blksz_min = (1 << 10);  /* 1K */
5047 /*
5048  * If set to less than the file block size, allow loaning out of an
5049  * arcbuf for a partial block read.  This must be a power of 2.
5050  */
5051 int zcr_blksz_max = (1 << 17);  /* 128K */

5053 /*ARGSUSED*/
5054 static int
5055 zfs_reqzcbuf(vnode_t *vp, enum uio_rw ioflag, xuio_t *xuio, cred_t *cr,
5056     caller_context_t *ct)
5057 {
```

```
5058         znode_t *zp = VTOZ(vp);
5059         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
5060         int max_blksz = zfsvfs->z_max_blksz;
5061         uio_t *uio = &xuio->xu_uio;
5062         ssize_t size = uio->uio_resid;
5063         offset_t offset = uio->uio_loffset;
5064         int blksz;
5065         int fullblk, i;
5066         arc_buf_t *abuf;
5067         ssize_t maxsize;
5068         int preamble, postamble;

5070         if (xuio->xu_type != UIOTYPE_ZEROCOPY)
5071                 return (SET_ERROR(EINVAL));

5073         ZFS_ENTER(zfsvfs);
5074         ZFS_VERIFY_ZP(zp);
5075         switch (ioflag) {
5076         case UIO_WRITE:
5077                 /*
5078                  * Loan out an arc_buf for write if write size is bigger than
5079                  * max_blksz, and the file's block size is also max_blksz.
5080                  */
5081                 blksz = max_blksz;
5082                 if (size < blksz || zp->z_blksz != blksz) {
5083                         ZFS_EXIT(zfsvfs);
5084                         return (SET_ERROR(EINVAL));
5085                 }
5086                 /*
5087                  * Caller requests buffers for write before knowing where the
5088                  * write offset might be (e.g. NFS TCP write).
5089                  */
5090                 if (offset == -1) {
5091                         preamble = 0;
5092                 } else {
5093                         preamble = P2PHASE(offset, blksz);
5094                         if (preamble) {
5095                                 preamble = blksz - preamble;
5096                                 size -= preamble;
5097                         }
5098                 }

5100                 postamble = P2PHASE(size, blksz);
5101                 size -= postamble;

5103                 fullblk = size / blksz;
5104                 (void) dmu_xuio_init(xuio,
5105                     (preamble != 0) + fullblk + (postamble != 0));
5106                 DTRACE_PROBE3(zfs_reqzcbuf_align, int, preamble,
5107                     int, postamble, int,
5108                     (preamble != 0) + fullblk + (postamble != 0));

5110                 /*
5111                  * Have to fix iov base/len for partial buffers.  They
5112                  * currently represent full arc_buf's.
5113                  */
5114                 if (preamble) {
5115                         /* data begins in the middle of the arc_buf */
5116                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5117                             blksz);
5118                         ASSERT(abuf);
5119                         (void) dmu_xuio_add(xuio, abuf,
5120                             blksz - preamble, preamble);
5121                 }

5123                 for (i = 0; i < fullblk; i++) {
```



```
5124                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5125                             blksz);
5126                         ASSERT(abuf);
5127                         (void) dmu_xuio_add(xuio, abuf, 0, blksz);
5128                 }

5130                 if (postamble) {
5131                         /* data ends in the middle of the arc_buf */
5132                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5133                             blksz);
5134                         ASSERT(abuf);
5135                         (void) dmu_xuio_add(xuio, abuf, 0, postamble);
5136                 }
5137                 break;
5138         case UIO_READ:
5139                 /*
5140                  * Loan out an arc_buf for read if the read size is larger than
5141                  * the current file block size.  Block alignment is not
5142                  * considered.  Partial arc_buf will be loaned out for read.
5143                  */
5144                 blksz = zp->z_blksz;
5145                 if (blksz < zcr_blksz_min)
5146                         blksz = zcr_blksz_min;
5147                 if (blksz > zcr_blksz_max)
5148                         blksz = zcr_blksz_max;
5149                 /* avoid potential complexity of dealing with it */
5150                 if (blksz > max_blksz) {
5151                         ZFS_EXIT(zfsvfs);
5152                         return (SET_ERROR(EINVAL));
5153                 }

5155                 maxsize = zp->z_size - uio->uio_loffset;
5156                 if (size > maxsize)
5157                         size = maxsize;

5159                 if (size < blksz || vn_has_cached_data(vp)) {
5160                         ZFS_EXIT(zfsvfs);
5161                         return (SET_ERROR(EINVAL));
5162                 }
5163                 break;
5164         default:
5165                 ZFS_EXIT(zfsvfs);
5166                 return (SET_ERROR(EINVAL));
5167         }

5169         uio->uio_extflg = UIO_XUIO;
5170         XUIO_XUZC_RW(xuio) = ioflag;
5171         ZFS_EXIT(zfsvfs);
5172         return (0);
5173 }

5175 /*ARGSUSED*/
5176 static int
5177 zfs_retzcbuf(vnode_t *vp, xuio_t *xuio, cred_t *cr, caller_context_t *ct)
5178 {
5179         int i;
5180         arc_buf_t *abuf;
5181         int ioflag = XUIO_XUZC_RW(xuio);

5183         ASSERT(xuio->xu_type == UIOTYPE_ZEROCOPY);

5185         i = dmu_xuio_cnt(xuio);
5186         while (i-- > 0) {
5187                 abuf = dmu_xuio_arcbuf(xuio, i);
5188                 /*
5189                  * if abuf == NULL, it must be a write buffer
```

```
5190                 * that has been returned in zfs_write().
5191                 */
5192                if (abuf)
5193                        dmu_return_arcbuf(abuf);
5194                ASSERT(abuf || ioflag == UIO_WRITE);
5195        }

5197        dmu_xuio_fini(xuio);
5198        return (0);
5199 }

5201 /*
5202  * Predeclare these here so that the compiler assumes that
5203  * this is an "old style" function declaration that does
5204  * not include arguments => we won't get type mismatch errors
5205  * in the initializations that follow.
5206  */
5207 static int zfs_inval();
5208 static int zfs_isdir();

5210 static int
5211 zfs_inval()
5212 {
5213        return (SET_ERROR(EINVAL));
5214 }

5216 static int
5217 zfs_isdir()
5218 {
5219        return (SET_ERROR(EISDIR));
5220 }
5221 /*
5222  * Directory vnode operations template
5223  */
5224 vnodeops_t *zfs_dvnodeops;
5225 const fs_operation_def_t zfs_dvnodeops_template[] = {
5226        VOPNAME_OPEN,           { .vop_open = zfs_open },
5227        VOPNAME_CLOSE,          { .vop_close = zfs_close },
5228        VOPNAME_READ,           { .error = zfs_isdir },
5229        VOPNAME_WRITE,          { .error = zfs_isdir },
5230        VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5231        VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5232        VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5233        VOPNAME_ACCESS,         { .vop_access = zfs_access },
5234        VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5235        VOPNAME_CREATE,         { .vop_create = zfs_create },
5236        VOPNAME_REMOVE,         { .vop_remove = zfs_remove },
5237        VOPNAME_LINK,           { .vop_link = zfs_link },
5238        VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5239        VOPNAME_MKDIR,          { .vop_mkdir = zfs_mkdir },
5240        VOPNAME_RMDIR,          { .vop_rmdir = zfs_rmdir },
5241        VOPNAME_READDIR,        { .vop_readdir = zfs_readdir },
5242        VOPNAME_SYMLINK,        { .vop_symlink = zfs_symlink },
5243        VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5244        VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5245        VOPNAME_FID,            { .vop_fid = zfs_fid },
5246        VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5247        VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5248        VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5249        VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5250        VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5251        NULL,                   NULL
5252 };

5254 /*
5255  * Regular file vnode operations template
```



```
5256  */
5257 vnodeops_t *zfs_fvnodeops;
5258 const fs_operation_def_t zfs_fvnodeops_template[] = {
5259        VOPNAME_OPEN,           { .vop_open = zfs_open },
5260        VOPNAME_CLOSE,          { .vop_close = zfs_close },
5261        VOPNAME_READ,           { .vop_read = zfs_read },
5262        VOPNAME_WRITE,          { .vop_write = zfs_write },
5263        VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5264        VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5265        VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5266        VOPNAME_ACCESS,         { .vop_access = zfs_access },
5267        VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5268        VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5269        VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5270        VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5271        VOPNAME_FID,            { .vop_fid = zfs_fid },
5272        VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5273        VOPNAME_FRLOCK,         { .vop_frlock = zfs_frlock },
5274        VOPNAME_SPACE,          { .vop_space = zfs_space },
5275        VOPNAME_GETPAGE,        { .vop_getpage = zfs_getpage },
5276        VOPNAME_PUTPAGE,        { .vop_putpage = zfs_putpage },
5277        VOPNAME_MAP,            { .vop_map = zfs_map },
5278        VOPNAME_ADDMAP,         { .vop_addmap = zfs_addmap },
5279        VOPNAME_DELMAP,         { .vop_delmap = zfs_delmap },
5280        VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5281        VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5282        VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5283        VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5284        VOPNAME_REQZCBUF,       { .vop_reqzcbuf = zfs_reqzcbuf },
5285        VOPNAME_RETZCBUF,       { .vop_retzcbuf = zfs_retzcbuf },
5286        NULL,                   NULL
5287 };

5289 /*
5290  * Symbolic link vnode operations template
5291  */
5292 vnodeops_t *zfs_symvnodeops;
5293 const fs_operation_def_t zfs_symvnodeops_template[] = {
5294        VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5295        VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5296        VOPNAME_ACCESS,         { .vop_access = zfs_access },
5297        VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5298        VOPNAME_READLINK,       { .vop_readlink = zfs_readlink },
5299        VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5300        VOPNAME_FID,            { .vop_fid = zfs_fid },
5301        VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5302        VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5303        NULL,                   NULL
5304 };

5306 /*
5307  * special share hidden files vnode operations template
5308  */
5309 vnodeops_t *zfs_sharevnodeops;
5310 const fs_operation_def_t zfs_sharevnodeops_template[] = {
5311        VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5312        VOPNAME_ACCESS,         { .vop_access = zfs_access },
5313        VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5314        VOPNAME_FID,            { .vop_fid = zfs_fid },
5315        VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5316        VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5317        VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5318        VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5319        NULL,                   NULL
5320 };
```

```
5322 /*
5323  * Extended attribute directory vnode operations template
5324  *
5325  * This template is identical to the directory vnodes
5326  * operation template except for restricted operations:
5327  *      VOP_MKDIR()
5328  *      VOP_SYMLINK()
5329  *
5330  * Note that there are other restrictions embedded in:
5331  *      zfs_create()    - restrict type to VREG
5332  *      zfs_link()      - no links into/out of attribute space
5333  *      zfs_rename()    - no moves into/out of attribute space
5334  */
5335 vnodeops_t *zfs_xdvnodeops;
5336 const fs_operation_def_t zfs_xdvnodeops_template[] = {
5337         VOPNAME_OPEN,           { .vop_open = zfs_open },
5338         VOPNAME_CLOSE,          { .vop_close = zfs_close },
5339         VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5340         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5341         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5342         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5343         VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5344         VOPNAME_CREATE,         { .vop_create = zfs_create },
5345         VOPNAME_REMOVE,         { .vop_remove = zfs_remove },
5346         VOPNAME_LINK,           { .vop_link = zfs_link },
5347         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5348         VOPNAME_MKDIR,          { .error = zfs_inval },
5349         VOPNAME_RMDIR,          { .vop_rmdir = zfs_rmdir },
5350         VOPNAME_READDIR,        { .vop_readdir = zfs_readdir },
5351         VOPNAME_SYMLINK,        { .error = zfs_inval },
5352         VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5353         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5354         VOPNAME_FID,            { .vop_fid = zfs_fid },
5355         VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5356         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5357         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5358         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5359         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5360         NULL,                   NULL
5361 };

5363 /*
5364  * Error vnode operations template
5365  */
5366 vnodeops_t *zfs_evnodeops;
5367 const fs_operation_def_t zfs_evnodeops_template[] = {
5368         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5369         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5370         NULL,                   NULL
5371 };
```