```
**********************************************************
   29782 Fri Nov 22 15:15:16 2013
new/usr/src/uts/common/fs/zfs/zfs_dir.c
4347 ZPL can use dmu_tx_assign(TXG_WAIT)
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 916 int
 917 zfs_make_xattrdir(znode_t *zp, vattr_t *vap, vnode_t **xvpp, cred_t *cr)
 918 {
 919         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
 920         znode_t *xzp;
 921         dmu_tx_t *tx;
 922         int error;
 923         zfs_acl_ids_t acl_ids;
 924         boolean_t fuid_dirtied;
 925         uint64_t parent;

 927         *xvpp = NULL;

 929         if (error = zfs_zaccess(zp, ACE_WRITE_NAMED_ATTRS, 0, B_FALSE, cr))
 930                 return (error);

 932         if ((error = zfs_acl_ids_create(zp, IS_XATTR, vap, cr, NULL,
 933             &acl_ids)) != 0)
 934                 return (error);
 935         if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
 936                 zfs_acl_ids_free(&acl_ids);
 937                 return (SET_ERROR(EDQUOT));
 938         }

 940 top:
 940         tx = dmu_tx_create(zfsvfs->z_os);
 941         dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
 942             ZFS_SA_BASE_ATTR_SIZE);
 943         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
 944         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
 945         fuid_dirtied = zfsvfs->z_fuid_dirty;
 946         if (fuid_dirtied)
 947                 zfs_fuid_txhold(zfsvfs, tx);
 948         error = dmu_tx_assign(tx, TXG_WAIT);
 949         error = dmu_tx_assign(tx, TXG_NOWAIT);
 949         if (error) {
 951                 if (error == ERESTART) {
 952                         dmu_tx_wait(tx);
 953                         dmu_tx_abort(tx);
 954                         goto top;
 955                 }
 950                 zfs_acl_ids_free(&acl_ids);
 951                 dmu_tx_abort(tx);
 952                 return (error);
 953         }
 954         zfs_mknode(zp, vap, tx, cr, IS_XATTR, &xzp, &acl_ids);

 956         if (fuid_dirtied)
 957                 zfs_fuid_sync(zfsvfs, tx);

 959 #ifdef DEBUG
 960         error = sa_lookup(xzp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
 961             &parent, sizeof (parent));
 962         ASSERT(error == 0 && parent == zp->z_id);
 963 #endif

 965         VERIFY(0 == sa_update(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs), &xzp->z_id,
```



```
 966             sizeof (xzp->z_id), tx));

 968         (void) zfs_log_create(zfsvfs->z_log, tx, TX_MKXATTR, zp,
 969             xzp, "", NULL, acl_ids.z_fuidp, vap);

 971         zfs_acl_ids_free(&acl_ids);
 972         dmu_tx_commit(tx);

 974         *xvpp = ZTOV(xzp);

 976         return (0);
 977 }
______unchanged_portion_omitted_
```

```
**********************************************************
  131808 Fri Nov 22 15:15:17 2013
new/usr/src/uts/common/fs/zfs/zfs_vnops.c
4347 ZPL can use dmu_tx_assign(TXG_WAIT)
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2013 by Delphix. All rights reserved.
  24  * Copyright 2013 Nexenta Systems, Inc.  All rights reserved.
  25  */

  27 /* Portions Copyright 2007 Jeremy Teo */
  28 /* Portions Copyright 2010 Robert Milkowski */

  30 #include <sys/types.h>
  31 #include <sys/param.h>
  32 #include <sys/time.h>
  33 #include <sys/systm.h>
  34 #include <sys/sysmacros.h>
  35 #include <sys/resource.h>
  36 #include <sys/vfs.h>
  37 #include <sys/vfs_opreg.h>
  38 #include <sys/vnode.h>
  39 #include <sys/file.h>
  40 #include <sys/stat.h>
  41 #include <sys/kmem.h>
  42 #include <sys/taskq.h>
  43 #include <sys/uio.h>
  44 #include <sys/vmsystm.h>
  45 #include <sys/atomic.h>
  46 #include <sys/vm.h>
  47 #include <vm/seg_vn.h>
  48 #include <vm/pvn.h>
  49 #include <vm/as.h>
  50 #include <vm/kpm.h>
  51 #include <vm/seg_kpm.h>
  52 #include <sys/mman.h>
  53 #include <sys/pathname.h>
  54 #include <sys/cmn_err.h>
  55 #include <sys/errno.h>
  56 #include <sys/unistd.h>
  57 #include <sys/zfs_dir.h>
  58 #include <sys/zfs_acl.h>
  59 #include <sys/zfs_ioctl.h>
```



```
  60 #include <sys/fs/zfs.h>
  61 #include <sys/dmu.h>
  62 #include <sys/dmu_objset.h>
  63 #include <sys/spa.h>
  64 #include <sys/txg.h>
  65 #include <sys/dbuf.h>
  66 #include <sys/zap.h>
  67 #include <sys/sa.h>
  68 #include <sys/dirent.h>
  69 #include <sys/policy.h>
  70 #include <sys/sunddi.h>
  71 #include <sys/filio.h>
  72 #include <sys/sid.h>
  73 #include "fs/fs_subr.h"
  74 #include <sys/zfs_ctldir.h>
  75 #include <sys/zfs_fuid.h>
  76 #include <sys/zfs_sa.h>
  77 #include <sys/dnlc.h>
  78 #include <sys/zfs_rlock.h>
  79 #include <sys/extdirent.h>
  80 #include <sys/kidmap.h>
  81 #include <sys/cred.h>
  82 #include <sys/attr.h>

  84 /*
  85  * Programming rules.
  86  *
  87  * Each vnode op performs some logical unit of work.  To do this, the ZPL must
  88  * properly lock its in-core state, create a DMU transaction, do the work,
  89  * record this work in the intent log (ZIL), commit the DMU transaction,
  90  * and wait for the intent log to commit if it is a synchronous operation.
  91  * Moreover, the vnode ops must work in both normal and log replay context.
  92  * The ordering of events is important to avoid deadlocks and references
  93  * to freed memory.  The example below illustrates the following Big Rules:
  94  *
  95  *  (1) A check must be made in each zfs thread for a mounted file system.
  96  *      This is done avoiding races using ZFS_ENTER(zfsvfs).
  97  *      A ZFS_EXIT(zfsvfs) is needed before all returns.  Any znodes
  98  *      must be checked with ZFS_VERIFY_ZP(zp).  Both of these macros
  99  *      can return EIO from the calling function.
 100  *
 101  *  (2) VN_RELE() should always be the last thing except for zil_commit()
 102  *      (if necessary) and ZFS_EXIT(). This is for 3 reasons:
 103  *      First, if it's the last reference, the vnode/znode
 104  *      can be freed, so the zp may point to freed memory.  Second, the last
 105  *      reference will call zfs_zinactive(), which may induce a lot of work --
 106  *      pushing cached pages (which acquires range locks) and syncing out
 107  *      cached atime changes.  Third, zfs_zinactive() may require a new tx,
 108  *      which could deadlock the system if you were already holding one.
 109  *      If you must call VN_RELE() within a tx then use VN_RELE_ASYNC().
 110  *
 111  *  (3) All range locks must be grabbed before calling dmu_tx_assign(),
 112  *      as they can span dmu_tx_assign() calls.
 113  *
 114  *  (4) If ZPL locks are held, pass TXG_NOWAIT as the second argument to
 115  *      dmu_tx_assign().  This is critical because we don't want to block
 116  *      while holding locks.
 117  *
 118  *      If no ZPL locks are held (aside from ZFS_ENTER()), use TXG_WAIT.  This
 119  *      reduces lock contention and CPU usage when we must wait (note that if
 120  *      throughput is constrained by the storage, nearly every transaction
 121  *      must wait).
 122  *
 114  *  (4) Always pass TXG_NOWAIT as the second argument to dmu_tx_assign().
 115  *      This is critical because we don't want to block while holding locks.
 123  *      Note, in particular, that if a lock is sometimes acquired before
```

```
 124  *      the tx assigns, and sometimes after (e.g. z_lock), then failing
 125  *      to use a non-blocking assign can deadlock the system.  The scenario:
 117  *      the tx assigns, and sometimes after (e.g. z_lock), then failing to
 118  *      use a non-blocking assign can deadlock the system.  The scenario:
 126  *
 127  *      Thread A has grabbed a lock before calling dmu_tx_assign().
 128  *      Thread B is in an already-assigned tx, and blocks for this lock.
 129  *      Thread A calls dmu_tx_assign(TXG_WAIT) and blocks in txg_wait_open()
 130  *      forever, because the previous txg can't quiesce until B's tx commits.
 131  *
 132  *      If dmu_tx_assign() returns ERESTART and zfsvfs->z_assign is TXG_NOWAIT,
 133  *      then drop all locks, call dmu_tx_wait(), and try again.  On subsequent
 134  *      calls to dmu_tx_assign(), pass TXG_WAITED rather than TXG_NOWAIT,
 135  *      to indicate that this operation has already called dmu_tx_wait().
 136  *      This will ensure that we don't retry forever, waiting a short bit
 137  *      each time.
 138  *
 139  *  (5) If the operation succeeded, generate the intent log entry for it
 140  *      before dropping locks.  This ensures that the ordering of events
 141  *      in the intent log matches the order in which they actually occurred.
 142  *      During ZIL replay the zfs_log_* functions will update the sequence
 143  *      number to indicate the zil transaction has replayed.
 144  *
 145  *  (6) At the end of each vnode op, the DMU tx must always commit,
 146  *      regardless of whether there were any errors.
 147  *
 148  *  (7) After dropping all locks, invoke zil_commit(zilog, foid)
 149  *      to ensure that synchronous semantics are provided when necessary.
 150  *
 151  * In general, this is how things should be ordered in each vnode op:
 152  *
 153  *      ZFS_ENTER(zfsvfs);              // exit if unmounted
 154  * top:
 155  *      zfs_dirent_lock(&dl, ...)       // lock directory entry (may VN_HOLD())
 156  *      rw_enter(...);                  // grab any other locks you need
 157  *      tx = dmu_tx_create(...);        // get DMU tx
 158  *      dmu_tx_hold_*();                // hold each object you might modify
 159  *      error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
 160  *      if (error) {
 161  *              rw_exit(...);           // drop locks
 162  *              zfs_dirent_unlock(dl);  // unlock directory entry
 163  *              VN_RELE(...);           // release held vnodes
 164  *              if (error == ERESTART) {
 165  *                      waited = B_TRUE;
 166  *                      dmu_tx_wait(tx);
 167  *                      dmu_tx_abort(tx);
 168  *                      goto top;
 169  *              }
 170  *              dmu_tx_abort(tx);       // abort DMU tx
 171  *              ZFS_EXIT(zfsvfs);       // finished in zfs
 172  *              return (error);         // really out of space
 173  *      }
 174  *      error = do_real_work();         // do whatever this VOP does
 175  *      if (error == 0)
 176  *              zfs_log_*(...);         // on success, make ZIL entry
 177  *      dmu_tx_commit(tx);              // commit DMU tx -- error or not
 178  *      rw_exit(...);                   // drop locks
 179  *      zfs_dirent_unlock(dl);          // unlock directory entry
 180  *      VN_RELE(...);                   // release held vnodes
 181  *      zil_commit(zilog, foid);        // synchronous when necessary
 182  *      ZFS_EXIT(zfsvfs);               // finished in zfs
 183  *      return (error);                 // done, report error
 184  */

 186 /* ARGSUSED */
 187 static int
```



```
 188 zfs_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
 189 {
 190         znode_t *zp = VTOZ(*vpp);
 191         zfsvfs_t *zfsvfs = zp->z_zfsvfs;

 193         ZFS_ENTER(zfsvfs);
 194         ZFS_VERIFY_ZP(zp);

 196         if ((flag & FWRITE) && (zp->z_pflags & ZFS_APPENDONLY) &&
 197             ((flag & FAPPEND) == 0)) {
 198                 ZFS_EXIT(zfsvfs);
 199                 return (SET_ERROR(EPERM));
 200         }

 202         if (!zfs_has_ctldir(zp) && zp->z_zfsvfs->z_vscan &&
 203             ZTOV(zp)->v_type == VREG &&
 204             !(zp->z_pflags & ZFS_AV_QUARANTINED) && zp->z_size > 0) {
 205                 if (fs_vscan(*vpp, cr, 0) != 0) {
 206                         ZFS_EXIT(zfsvfs);
 207                         return (SET_ERROR(EACCES));
 208                 }
 209         }

 211         /* Keep a count of the synchronous opens in the znode */
 212         if (flag & (FSYNC | FDSYNC))
 213                 atomic_inc_32(&zp->z_sync_cnt);

 215         ZFS_EXIT(zfsvfs);
 216         return (0);
 217 }
```

*unchanged_portion_omitted_*

```
 579 /*
 580  * Write the bytes to a file.
 581  *
 582  *      IN:     vp      - vnode of file to be written to.
 583  *              uio     - structure supplying write location, range info,
 584  *                        and data buffer.
 585  *              ioflag  - FAPPEND, FSYNC, and/or FDSYNC.  FAPPEND is
 586  *                        set if in append mode.
 587  *              cr      - credentials of caller.
 588  *              ct      - caller context (NFS/CIFS fem monitor only)
 589  *
 590  *      OUT:    uio     - updated offset and range.
 591  *
 592  *      RETURN: 0 on success, error code on failure.
 593  *
 594  * Timestamps:
 595  *      vp - ctime|mtime updated if byte count > 0
 596  */

 598 /* ARGSUSED */
 599 static int
 600 zfs_write(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cr, caller_context_t *ct)
 601 {
 602         znode_t         *zp = VTOZ(vp);
 603         rlim64_t        limit = uio->uio_llimit;
 604         ssize_t         start_resid = uio->uio_resid;
 605         ssize_t         tx_bytes;
 606         uint64_t        end_size;
 607         dmu_tx_t        *tx;
 608         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
 609         zilog_t         *zilog;
 610         offset_t        woff;
 611         ssize_t         n, nbytes;
 612         rl_t            *rl;
```

```
613         int             max_blksz = zfsvfs->z_max_blksz;
614         int             error = 0;
615         arc_buf_t       *abuf;
616         iovec_t         *aiov = NULL;
617         xuio_t          *xuio = NULL;
618         int             i_iov = 0;
619         int             iovcnt = uio->uio_iovcnt;
620         iovec_t         *iovp = uio->uio_iov;
621         int             write_eof;
622         int             count = 0;
623         sa_bulk_attr_t  bulk[4];
624         uint64_t        mtime[2], ctime[2];

626         /*
627          * Fasttrack empty write
628          */
629         n = start_resid;
630         if (n == 0)
631                 return (0);

633         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
634                 limit = MAXOFFSET_T;

636         ZFS_ENTER(zfsvfs);
637         ZFS_VERIFY_ZP(zp);

639         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
640         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);
641         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_SIZE(zfsvfs), NULL,
642             &zp->z_size, 8);
643         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
644             &zp->z_pflags, 8);

646         /*
647          * If immutable or not appending then return EPERM
648          */
649         if ((zp->z_pflags & (ZFS_IMMUTABLE | ZFS_READONLY)) ||
650             ((zp->z_pflags & ZFS_APPENDONLY) && !(ioflag & FAPPEND) &&
651             (uio->uio_loffset < zp->z_size))) {
652                 ZFS_EXIT(zfsvfs);
653                 return (SET_ERROR(EPERM));
654         }

656         zilog = zfsvfs->z_log;

658         /*
659          * Validate file offset
660          */
661         woff = ioflag & FAPPEND ? zp->z_size : uio->uio_loffset;
662         if (woff < 0) {
663                 ZFS_EXIT(zfsvfs);
664                 return (SET_ERROR(EINVAL));
665         }

667         /*
668          * Check for mandatory locks before calling zfs_range_lock()
669          * in order to prevent a deadlock with locks set via fcntl().
670          */
671         if (MANDMODE((mode_t)zp->z_mode) &&
672             (error = chklock(vp, FWRITE, woff, n, uio->uio_fmode, ct)) != 0) {
673                 ZFS_EXIT(zfsvfs);
674                 return (error);
675         }

677         /*
678          * Pre-fault the pages to ensure slow (eg NFS) pages
```



```
679          * don't hold up txg.
680          * Skip this if uio contains loaned arc_buf.
681          */
682         if ((uio->uio_extflg == UIO_XUIO) &&
683             (((xuio_t *)uio)->xu_type == UIOTYPE_ZEROCOPY))
684                 xuio = (xuio_t *)uio;
685         else
686                 uio_prefaultpages(MIN(n, max_blksz), uio);

688         /*
689          * If in append mode, set the io offset pointer to eof.
690          */
691         if (ioflag & FAPPEND) {
692                 /*
693                  * Obtain an appending range lock to guarantee file append
694                  * semantics.  We reset the write offset once we have the lock.
695                  */
696                 rl = zfs_range_lock(zp, 0, n, RL_APPEND);
697                 woff = rl->r_off;
698                 if (rl->r_len == UINT64_MAX) {
699                         /*
700                          * We overlocked the file because this write will cause
701                          * the file block size to increase.
702                          * Note that zp_size cannot change with this lock held.
703                          */
704                         woff = zp->z_size;
705                 }
706                 uio->uio_loffset = woff;
707         } else {
708                 /*
709                  * Note that if the file block size will change as a result of
710                  * this write, then this range lock will lock the entire file
711                  * so that we can re-write the block safely.
712                  */
713                 rl = zfs_range_lock(zp, woff, n, RL_WRITER);
714         }

716         if (woff >= limit) {
717                 zfs_range_unlock(rl);
718                 ZFS_EXIT(zfsvfs);
719                 return (SET_ERROR(EFBIG));
720         }

722         if ((woff + n) > limit || woff > (limit - n))
723                 n = limit - woff;

725         /* Will this write extend the file length? */
726         write_eof = (woff + n > zp->z_size);

728         end_size = MAX(zp->z_size, woff + n);

730         /*
731          * Write the file in reasonable size chunks.  Each chunk is written
732          * in a separate transaction; this keeps the intent log records small
733          * and allows us to do more fine-grained space accounting.
734          */
735         while (n > 0) {
736                 abuf = NULL;
737                 woff = uio->uio_loffset;
731 again:
738                 if (zfs_owner_overquota(zfsvfs, zp, B_FALSE) ||
739                     zfs_owner_overquota(zfsvfs, zp, B_TRUE)) {
740                         if (abuf != NULL)
741                                 dmu_return_arcbuf(abuf);
742                         error = SET_ERROR(EDQUOT);
743                         break;
```

```
744                 }
745
746                 if (xuio && abuf == NULL) {
747                         ASSERT(i_iov < iovcnt);
748                         aiov = &iovp[i_iov];
749                         abuf = dmu_xuio_arcbuf(xuio, i_iov);
750                         dmu_xuio_clear(xuio, i_iov);
751                         DTRACE_PROBE3(zfs_cp_write, int, i_iov,
752                             iovec_t *, aiov, arc_buf_t *, abuf);
753                         ASSERT((aiov->iov_base == abuf->b_data) ||
754                             ((char *)aiov->iov_base - (char *)abuf->b_data +
755                             aiov->iov_len == arc_buf_size(abuf)));
756                         i_iov++;
757                 } else if (abuf == NULL && n >= max_blksz &&
758                     woff >= zp->z_size &&
759                     P2PHASE(woff, max_blksz) == 0 &&
760                     zp->z_blksz == max_blksz) {
761                         /*
762                          * This write covers a full block.  "Borrow" a buffer
763                          * from the dmu so that we can fill it before we enter
764                          * a transaction.  This avoids the possibility of
765                          * holding up the transaction if the data copy hangs
766                          * up on a pagefault (e.g., from an NFS server mapping).
767                          */
768                         size_t cbytes;
769
770                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
771                             max_blksz);
772                         ASSERT(abuf != NULL);
773                         ASSERT(arc_buf_size(abuf) == max_blksz);
774                         if (error = uiocopy(abuf->b_data, max_blksz,
775                             UIO_WRITE, uio, &cbytes)) {
776                                 dmu_return_arcbuf(abuf);
777                                 break;
778                         }
779                         ASSERT(cbytes == max_blksz);
780                 }
781
782                 /*
783                  * Start a transaction.
784                  */
785                 tx = dmu_tx_create(zfsvfs->z_os);
786                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
787                 dmu_tx_hold_write(tx, zp->z_id, woff, MIN(n, max_blksz));
788                 zfs_sa_upgrade_txholds(tx, zp);
789                 error = dmu_tx_assign(tx, TXG_WAIT);
783                 error = dmu_tx_assign(tx, TXG_NOWAIT);
790                 if (error) {
785                         if (error == ERESTART) {
786                                 dmu_tx_wait(tx);
791                         dmu_tx_abort(tx);
788                                 goto again;
789                         }
790                         dmu_tx_abort(tx);
792                         if (abuf != NULL)
793                                 dmu_return_arcbuf(abuf);
794                         break;
795                 }
796
797                 /*
798                  * If zfs_range_lock() over-locked we grow the blocksize
799                  * and then reduce the lock range.  This will only happen
800                  * on the first iteration since zfs_range_reduce() will
801                  * shrink down r_len to the appropriate size.
802                  */
803                 if (rl->r_len == UINT64_MAX) {
```



```
804                         uint64_t new_blksz;
805
806                         if (zp->z_blksz > max_blksz) {
807                                 ASSERT(!ISP2(zp->z_blksz));
808                                 new_blksz = MIN(end_size, SPA_MAXBLOCKSIZE);
809                         } else {
810                                 new_blksz = MIN(end_size, max_blksz);
811                         }
812                         zfs_grow_blocksize(zp, new_blksz, tx);
813                         zfs_range_reduce(rl, woff, n);
814                 }
815
816                 /*
817                  * XXX - should we really limit each write to z_max_blksz?
818                  * Perhaps we should use SPA_MAXBLOCKSIZE chunks?
819                  */
820                 nbytes = MIN(n, max_blksz - P2PHASE(woff, max_blksz));
821
822                 if (abuf == NULL) {
823                         tx_bytes = uio->uio_resid;
824                         error = dmu_write_uio_dbuf(sa_get_db(zp->z_sa_hdl),
825                             uio, nbytes, tx);
826                         tx_bytes -= uio->uio_resid;
827                 } else {
828                         tx_bytes = nbytes;
829                         ASSERT(xuio == NULL || tx_bytes == aiov->iov_len);
830                         /*
831                          * If this is not a full block write, but we are
832                          * extending the file past EOF and this data starts
833                          * block-aligned, use assign_arcbuf().  Otherwise,
834                          * write via dmu_write().
835                          */
836                         if (tx_bytes < max_blksz && (!write_eof ||
837                             aiov->iov_base != abuf->b_data)) {
838                                 ASSERT(xuio);
839                                 dmu_write(zfsvfs->z_os, zp->z_id, woff,
840                                     aiov->iov_len, aiov->iov_base, tx);
841                                 dmu_return_arcbuf(abuf);
842                                 xuio_stat_wbuf_copied();
843                         } else {
844                                 ASSERT(xuio || tx_bytes == max_blksz);
845                                 dmu_assign_arcbuf(sa_get_db(zp->z_sa_hdl),
846                                     woff, abuf, tx);
847                         }
848                         ASSERT(tx_bytes <= uio->uio_resid);
849                         uioskip(uio, tx_bytes);
850                 }
851                 if (tx_bytes && vn_has_cached_data(vp)) {
852                         update_pages(vp, woff,
853                             tx_bytes, zfsvfs->z_os, zp->z_id);
854                 }
855
856                 /*
857                  * If we made no progress, we're done.  If we made even
858                  * partial progress, update the znode and ZIL accordingly.
859                  */
860                 if (tx_bytes == 0) {
861                         (void) sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zfsvfs),
862                             (void *)&zp->z_size, sizeof (uint64_t), tx);
863                         dmu_tx_commit(tx);
864                         ASSERT(error != 0);
865                         break;
866                 }
867
868                 /*
869                  * Clear Set-UID/Set-GID bits on successful write if not
```

```
870                  * privileged and at least one of the excute bits is set.
871                  *
872                  * It would be nice to to this after all writes have
873                  * been done, but that would still expose the ISUID/ISGID
874                  * to another app after the partial write is committed.
875                  *
876                  * Note: we don't call zfs_fuid_map_id() here because
877                  * user 0 is not an ephemeral uid.
878                  */
879                 mutex_enter(&zp->z_acl_lock);
880                 if ((zp->z_mode & (S_IXUSR | (S_IXUSR >> 3) |
881                     (S_IXUSR >> 6))) != 0 &&
882                     (zp->z_mode & (S_ISUID | S_ISGID)) != 0 &&
883                     secpolicy_vnode_setid_retain(cr,
884                     (zp->z_mode & S_ISUID) != 0 && zp->z_uid == 0) != 0) {
885                         uint64_t newmode;
886                         zp->z_mode &= ~(S_ISUID | S_ISGID);
887                         newmode = zp->z_mode;
888                         (void) sa_update(zp->z_sa_hdl, SA_ZPL_MODE(zfsvfs),
889                             (void *)&newmode, sizeof (uint64_t), tx);
890                 }
891                 mutex_exit(&zp->z_acl_lock);

893                 zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
894                     B_TRUE);

896                 /*
897                  * Update the file size (zp_size) if it has changed;
898                  * account for possible concurrent updates.
899                  */
900                 while ((end_size = zp->z_size) < uio->uio_loffset) {
901                         (void) atomic_cas_64(&zp->z_size, end_size,
902                             uio->uio_loffset);
903                         ASSERT(error == 0);
904                 }
905                 /*
906                  * If we are replaying and eof is non zero then force
907                  * the file size to the specified eof. Note, there's no
908                  * concurrency during replay.
909                  */
910                 if (zfsvfs->z_replay && zfsvfs->z_replay_eof != 0)
911                         zp->z_size = zfsvfs->z_replay_eof;

913                 error = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);

915                 zfs_log_write(zilog, tx, TX_WRITE, zp, woff, tx_bytes, ioflag);
916                 dmu_tx_commit(tx);

918                 if (error != 0)
919                         break;
920                 ASSERT(tx_bytes == nbytes);
921                 n -= nbytes;

923                 if (!xuio && n > 0)
924                         uio_prefaultpages(MIN(n, max_blksz), uio);
925         }

927         zfs_range_unlock(rl);

929         /*
930          * If we're in replay mode, or we made no progress, return error.
931          * Otherwise, it's at least a partial write, so it's successful.
932          */
933         if (zfsvfs->z_replay || uio->uio_resid == start_resid) {
934                 ZFS_EXIT(zfsvfs);
935                 return (error);
```



```
936         }

938         if (ioflag & (FSYNC | FDSYNC) ||
939             zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
940                 zil_commit(zilog, zp->z_id);

942         ZFS_EXIT(zfsvfs);
943         return (0);
944 }
```

unchanged_portion_omitted

```
2610 /*
2611  * Set the file attributes to the values contained in the
2612  * vattr structure.
2613  *
2614  *      IN:     vp      - vnode of file to be modified.
2615  *              vap     - new attribute values.
2616  *                        If AT_XVATTR set, then optional attrs are being set
2617  *              flags   - ATTR_UTIME set if non-default time values provided.
2618  *                      - ATTR_NOACLCHECK (CIFS context only).
2619  *              cr      - credentials of caller.
2620  *              ct      - caller context
2621  *
2622  *      RETURN: 0 on success, error code on failure.
2623  *
2624  * Timestamps:
2625  *      vp - ctime updated, mtime updated if size changed.
2626  */
2627 /* ARGSUSED */
2628 static int
2629 zfs_setattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2630     caller_context_t *ct)
2631 {
2632         znode_t         *zp = VTOZ(vp);
2633         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2634         zilog_t         *zilog;
2635         dmu_tx_t        *tx;
2636         vattr_t         oldva;
2637         xvattr_t        tmpxvattr;
2638         uint_t          mask = vap->va_mask;
2639         uint_t          saved_mask = 0;
2640         int             trim_mask = 0;
2641         uint64_t        new_mode;
2642         uint64_t        new_uid, new_gid;
2643         uint64_t        xattr_obj;
2644         uint64_t        mtime[2], ctime[2];
2645         znode_t         *attrzp;
2646         int             need_policy = FALSE;
2647         int             err, err2;
2648         zfs_fuid_info_t *fuidp = NULL;
2649         xvattr_t *xvap = (xvattr_t *)vap;       /* vap may be an xvattr_t * */
2650         xoptattr_t      *xoap;
2651         zfs_acl_t       *aclp;
2652         boolean_t skipaclchk = (flags & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
2653         boolean_t       fuid_dirtied = B_FALSE;
2654         sa_bulk_attr_t  bulk[7], xattr_bulk[7];
2655         int             count = 0, xattr_count = 0;

2657         if (mask == 0)
2658                 return (0);

2660         if (mask & AT_NOSET)
2661                 return (SET_ERROR(EINVAL));

2663         ZFS_ENTER(zfsvfs);
2664         ZFS_VERIFY_ZP(zp);
```

```
2666         zilog = zfsvfs->z_log;

2668         /*
2669          * Make sure that if we have ephemeral uid/gid or xvattr specified
2670          * that file system is at proper version level
2671          */

2673         if (zfsvfs->z_use_fuids == B_FALSE &&
2674             (((mask & AT_UID) && IS_EPHEMERAL(vap->va_uid)) ||
2675             ((mask & AT_GID) && IS_EPHEMERAL(vap->va_gid)) ||
2676             (mask & AT_XVATTR))) {
2677                 ZFS_EXIT(zfsvfs);
2678                 return (SET_ERROR(EINVAL));
2679         }

2681         if (mask & AT_SIZE && vp->v_type == VDIR) {
2682                 ZFS_EXIT(zfsvfs);
2683                 return (SET_ERROR(EISDIR));
2684         }

2686         if (mask & AT_SIZE && vp->v_type != VREG && vp->v_type != VFIFO) {
2687                 ZFS_EXIT(zfsvfs);
2688                 return (SET_ERROR(EINVAL));
2689         }

2691         /*
2692          * If this is an xvattr_t, then get a pointer to the structure of
2693          * optional attributes.  If this is NULL, then we have a vattr_t.
2694          */
2695         xoap = xva_getxoptattr(xvap);

2697         xva_init(&tmpxvattr);

2699         /*
2700          * Immutable files can only alter immutable bit and atime
2701          */
2702         if ((zp->z_pflags & ZFS_IMMUTABLE) &&
2703             ((mask & (AT_SIZE|AT_UID|AT_GID|AT_MTIME|AT_MODE)) ||
2704             ((mask & AT_XVATTR) && XVA_ISSET_REQ(xvap, XAT_CREATETIME)))) {
2705                 ZFS_EXIT(zfsvfs);
2706                 return (SET_ERROR(EPERM));
2707         }

2709         if ((mask & AT_SIZE) && (zp->z_pflags & ZFS_READONLY)) {
2710                 ZFS_EXIT(zfsvfs);
2711                 return (SET_ERROR(EPERM));
2712         }

2714         /*
2715          * Verify timestamps doesn't overflow 32 bits.
2716          * ZFS can handle large timestamps, but 32bit syscalls can't
2717          * handle times greater than 2039.  This check should be removed
2718          * once large timestamps are fully supported.
2719          */
2720         if (mask & (AT_ATIME | AT_MTIME)) {
2721                 if (((mask & AT_ATIME) && TIMESPEC_OVERFLOW(&vap->va_atime)) ||
2722                     ((mask & AT_MTIME) && TIMESPEC_OVERFLOW(&vap->va_mtime))) {
2723                         ZFS_EXIT(zfsvfs);
2724                         return (SET_ERROR(EOVERFLOW));
2725                 }
2726         }

2728 top:
2729         attrzp = NULL;
2730         aclp = NULL;
```



```
2732         /* Can this be moved to before the top label? */
2733         if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
2734                 ZFS_EXIT(zfsvfs);
2735                 return (SET_ERROR(EROFS));
2736         }

2738         /*
2739          * First validate permissions
2740          */

2742         if (mask & AT_SIZE) {
2743                 err = zfs_zaccess(zp, ACE_WRITE_DATA, 0, skipaclchk, cr);
2744                 if (err) {
2745                         ZFS_EXIT(zfsvfs);
2746                         return (err);
2747                 }
2748                 /*
2749                  * XXX - Note, we are not providing any open
2750                  * mode flags here (like FNDELAY), so we may
2751                  * block if there are locks present... this
2752                  * should be addressed in openat().
2753                  */
2754                 /* XXX - would it be OK to generate a log record here? */
2755                 err = zfs_freesp(zp, vap->va_size, 0, 0, FALSE);
2756                 if (err) {
2757                         ZFS_EXIT(zfsvfs);
2758                         return (err);
2759                 }

2761                 if (vap->va_size == 0)
2762                         vnevent_truncate(ZTOV(zp), ct);
2763         }

2765         if (mask & (AT_ATIME|AT_MTIME) ||
2766             ((mask & AT_XVATTR) && (XVA_ISSET_REQ(xvap, XAT_HIDDEN) ||
2767             XVA_ISSET_REQ(xvap, XAT_READONLY) ||
2768             XVA_ISSET_REQ(xvap, XAT_ARCHIVE) ||
2769             XVA_ISSET_REQ(xvap, XAT_OFFLINE) ||
2770             XVA_ISSET_REQ(xvap, XAT_SPARSE) ||
2771             XVA_ISSET_REQ(xvap, XAT_CREATETIME) ||
2772             XVA_ISSET_REQ(xvap, XAT_SYSTEM)))) {
2773                 need_policy = zfs_zaccess(zp, ACE_WRITE_ATTRIBUTES, 0,
2774                     skipaclchk, cr);
2775         }

2777         if (mask & (AT_UID|AT_GID)) {
2778                 int     idmask = (mask & (AT_UID|AT_GID));
2779                 int     take_owner;
2780                 int     take_group;

2782                 /*
2783                  * NOTE: even if a new mode is being set,
2784                  * we may clear S_ISUID/S_ISGID bits.
2785                  */

2787                 if (!(mask & AT_MODE))
2788                         vap->va_mode = zp->z_mode;

2790                 /*
2791                  * Take ownership or chgrp to group we are a member of
2792                  */

2794                 take_owner = (mask & AT_UID) && (vap->va_uid == crgetuid(cr));
2795                 take_group = (mask & AT_GID) &&
2796                     zfs_groupmember(zfsvfs, vap->va_gid, cr);
```

```
2798                 /*
2799                  * If both AT_UID and AT_GID are set then take_owner and
2800                  * take_group must both be set in order to allow taking
2801                  * ownership.
2802                  *
2803                  * Otherwise, send the check through secpolicy_vnode_setattr()
2804                  *
2805                  */

2807                 if (((idmask == (AT_UID|AT_GID)) && take_owner && take_group) ||
2808                     ((idmask == AT_UID) && take_owner) ||
2809                     ((idmask == AT_GID) && take_group)) {
2810                         if (zfs_zaccess(zp, ACE_WRITE_OWNER, 0,
2811                             skipaclchk, cr) == 0) {
2812                                 /*
2813                                  * Remove setuid/setgid for non-privileged users
2814                                  */
2815                                 secpolicy_setid_clear(vap, cr);
2816                                 trim_mask = (mask & (AT_UID|AT_GID));
2817                         } else {
2818                                 need_policy =  TRUE;
2819                         }
2820                 } else {
2821                         need_policy =  TRUE;
2822                 }
2823         }

2825         mutex_enter(&zp->z_lock);
2826         oldva.va_mode = zp->z_mode;
2827         zfs_fuid_map_ids(zp, cr, &oldva.va_uid, &oldva.va_gid);
2828         if (mask & AT_XVATTR) {
2829                 /*
2830                  * Update xvattr mask to include only those attributes
2831                  * that are actually changing.
2832                  *
2833                  * the bits will be restored prior to actually setting
2834                  * the attributes so the caller thinks they were set.
2835                  */
2836                 if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2837                         if (xoap->xoa_appendonly !=
2838                             ((zp->z_pflags & ZFS_APPENDONLY) != 0)) {
2839                                 need_policy = TRUE;
2840                         } else {
2841                                 XVA_CLR_REQ(xvap, XAT_APPENDONLY);
2842                                 XVA_SET_REQ(&tmpxvattr, XAT_APPENDONLY);
2843                         }
2844                 }

2846                 if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2847                         if (xoap->xoa_nounlink !=
2848                             ((zp->z_pflags & ZFS_NOUNLINK) != 0)) {
2849                                 need_policy = TRUE;
2850                         } else {
2851                                 XVA_CLR_REQ(xvap, XAT_NOUNLINK);
2852                                 XVA_SET_REQ(&tmpxvattr, XAT_NOUNLINK);
2853                         }
2854                 }

2856                 if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2857                         if (xoap->xoa_immutable !=
2858                             ((zp->z_pflags & ZFS_IMMUTABLE) != 0)) {
2859                                 need_policy = TRUE;
2860                         } else {
2861                                 XVA_CLR_REQ(xvap, XAT_IMMUTABLE);
2862                                 XVA_SET_REQ(&tmpxvattr, XAT_IMMUTABLE);
```



```
2863                         }
2864                 }

2866                 if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2867                         if (xoap->xoa_nodump !=
2868                             ((zp->z_pflags & ZFS_NODUMP) != 0)) {
2869                                 need_policy = TRUE;
2870                         } else {
2871                                 XVA_CLR_REQ(xvap, XAT_NODUMP);
2872                                 XVA_SET_REQ(&tmpxvattr, XAT_NODUMP);
2873                         }
2874                 }

2876                 if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
2877                         if (xoap->xoa_av_modified !=
2878                             ((zp->z_pflags & ZFS_AV_MODIFIED) != 0)) {
2879                                 need_policy = TRUE;
2880                         } else {
2881                                 XVA_CLR_REQ(xvap, XAT_AV_MODIFIED);
2882                                 XVA_SET_REQ(&tmpxvattr, XAT_AV_MODIFIED);
2883                         }
2884                 }

2886                 if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
2887                         if ((vp->v_type != VREG &&
2888                             xoap->xoa_av_quarantined) ||
2889                             xoap->xoa_av_quarantined !=
2890                             ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0)) {
2891                                 need_policy = TRUE;
2892                         } else {
2893                                 XVA_CLR_REQ(xvap, XAT_AV_QUARANTINED);
2894                                 XVA_SET_REQ(&tmpxvattr, XAT_AV_QUARANTINED);
2895                         }
2896                 }

2898                 if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
2899                         mutex_exit(&zp->z_lock);
2900                         ZFS_EXIT(zfsvfs);
2901                         return (SET_ERROR(EPERM));
2902                 }

2904                 if (need_policy == FALSE &&
2905                     (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) ||
2906                     XVA_ISSET_REQ(xvap, XAT_OPAQUE))) {
2907                         need_policy = TRUE;
2908                 }
2909         }

2911         mutex_exit(&zp->z_lock);

2913         if (mask & AT_MODE) {
2914                 if (zfs_zaccess(zp, ACE_WRITE_ACL, 0, skipaclchk, cr) == 0) {
2915                         err = secpolicy_setid_setsticky_clear(vp, vap,
2916                             &oldva, cr);
2917                         if (err) {
2918                                 ZFS_EXIT(zfsvfs);
2919                                 return (err);
2920                         }
2921                         trim_mask |= AT_MODE;
2922                 } else {
2923                         need_policy = TRUE;
2924                 }
2925         }

2927         if (need_policy) {
2928                 /*
```

```
2929                  * If trim_mask is set then take ownership
2930                  * has been granted or write_acl is present and user
2931                  * has the ability to modify mode.  In that case remove
2932                  * UID|GID and or MODE from mask so that
2933                  * secpolicy_vnode_setattr() doesn't revoke it.
2934                  */

2936                 if (trim_mask) {
2937                         saved_mask = vap->va_mask;
2938                         vap->va_mask &= ~trim_mask;
2939                 }
2940                 err = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
2941                     (int (*)(void *, int, cred_t *))zfs_zaccess_unix, zp);
2942                 if (err) {
2943                         ZFS_EXIT(zfsvfs);
2944                         return (err);
2945                 }

2947                 if (trim_mask)
2948                         vap->va_mask |= saved_mask;
2949         }

2951         /*
2952          * secpolicy_vnode_setattr, or take ownership may have
2953          * changed va_mask
2954          */
2955         mask = vap->va_mask;

2957         if ((mask & (AT_UID | AT_GID))) {
2958                 err = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
2959                     &xattr_obj, sizeof (xattr_obj));

2961                 if (err == 0 && xattr_obj) {
2962                         err = zfs_zget(zp->z_zfsvfs, xattr_obj, &attrzp);
2963                         if (err)
2964                                 goto out2;
2965                 }
2966                 if (mask & AT_UID) {
2967                         new_uid = zfs_fuid_create(zfsvfs,
2968                             (uint64_t)vap->va_uid, cr, ZFS_OWNER, &fuidp);
2969                         if (new_uid != zp->z_uid &&
2970                             zfs_fuid_overquota(zfsvfs, B_FALSE, new_uid)) {
2971                                 if (attrzp)
2972                                         VN_RELE(ZTOV(attrzp));
2973                                 err = SET_ERROR(EDQUOT);
2974                                 goto out2;
2975                         }
2976                 }

2978                 if (mask & AT_GID) {
2979                         new_gid = zfs_fuid_create(zfsvfs, (uint64_t)vap->va_gid,
2980                             cr, ZFS_GROUP, &fuidp);
2981                         if (new_gid != zp->z_gid &&
2982                             zfs_fuid_overquota(zfsvfs, B_TRUE, new_gid)) {
2983                                 if (attrzp)
2984                                         VN_RELE(ZTOV(attrzp));
2985                                 err = SET_ERROR(EDQUOT);
2986                                 goto out2;
2987                         }
2988                 }
2989         }
2990         tx = dmu_tx_create(zfsvfs->z_os);

2992         if (mask & AT_MODE) {
2993                 uint64_t pmode = zp->z_mode;
2994                 uint64_t acl_obj;
```



```
2995                 new_mode = (pmode & S_IFMT) | (vap->va_mode & ~S_IFMT);

2997                 if (zp->z_zfsvfs->z_acl_mode == ZFS_ACL_RESTRICTED &&
2998                     !(zp->z_pflags & ZFS_ACL_TRIVIAL)) {
2999                         err = SET_ERROR(EPERM);
3000                         goto out;
3001                 }

3003                 if (err = zfs_acl_chmod_setattr(zp, &aclp, new_mode))
3004                         goto out;

3006                 mutex_enter(&zp->z_lock);
3007                 if (!zp->z_is_sa && ((acl_obj = zfs_external_acl(zp)) != 0)) {
3008                         /*
3009                          * Are we upgrading ACL from old V0 format
3010                          * to V1 format?
3011                          */
3012                         if (zfsvfs->z_version >= ZPL_VERSION_FUID &&
3013                             zfs_znode_acl_version(zp) ==
3014                             ZFS_ACL_VERSION_INITIAL) {
3015                                 dmu_tx_hold_free(tx, acl_obj, 0,
3016                                     DMU_OBJECT_END);
3017                                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3018                                     0, aclp->z_acl_bytes);
3019                         } else {
3020                                 dmu_tx_hold_write(tx, acl_obj, 0,
3021                                     aclp->z_acl_bytes);
3022                         }
3023                 } else if (!zp->z_is_sa && aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3024                         dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3025                             0, aclp->z_acl_bytes);
3026                 }
3027                 mutex_exit(&zp->z_lock);
3028                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3029         } else {
3030                 if ((mask & AT_XVATTR) &&
3031                     XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3032                         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3033                 else
3034                         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
3035         }

3037         if (attrzp) {
3038                 dmu_tx_hold_sa(tx, attrzp->z_sa_hdl, B_FALSE);
3039         }

3041         fuid_dirtied = zfsvfs->z_fuid_dirty;
3042         if (fuid_dirtied)
3043                 zfs_fuid_txhold(zfsvfs, tx);

3045         zfs_sa_upgrade_txholds(tx, zp);

3047         err = dmu_tx_assign(tx, TXG_WAIT);
3048         if (err)
3046         err = dmu_tx_assign(tx, TXG_NOWAIT);
3047         if (err) {
3048                 if (err == ERESTART)
3049                         dmu_tx_wait(tx);
3049                 goto out;
3051         }

3051         count = 0;
3052         /*
3053          * Set each attribute requested.
3054          * We group settings according to the locks they need to acquire.
3055          *
```

```
3056          * Note: you cannot set ctime directly, although it will be
3057          * updated as a side-effect of calling this function.
3058          */


3061         if (mask & (AT_UID|AT_GID|AT_MODE))
3062                 mutex_enter(&zp->z_acl_lock);
3063         mutex_enter(&zp->z_lock);

3065         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
3066             &zp->z_pflags, sizeof (zp->z_pflags));

3068         if (attrzp) {
3069                 if (mask & (AT_UID|AT_GID|AT_MODE))
3070                         mutex_enter(&attrzp->z_acl_lock);
3071                 mutex_enter(&attrzp->z_lock);
3072                 SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3073                     SA_ZPL_FLAGS(zfsvfs), NULL, &attrzp->z_pflags,
3074                     sizeof (attrzp->z_pflags));
3075         }

3077         if (mask & (AT_UID|AT_GID)) {

3079                 if (mask & AT_UID) {
3080                         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_UID(zfsvfs), NULL,
3081                             &new_uid, sizeof (new_uid));
3082                         zp->z_uid = new_uid;
3083                         if (attrzp) {
3084                                 SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3085                                     SA_ZPL_UID(zfsvfs), NULL, &new_uid,
3086                                     sizeof (new_uid));
3087                                 attrzp->z_uid = new_uid;
3088                         }
3089                 }

3091                 if (mask & AT_GID) {
3092                         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_GID(zfsvfs),
3093                             NULL, &new_gid, sizeof (new_gid));
3094                         zp->z_gid = new_gid;
3095                         if (attrzp) {
3096                                 SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3097                                     SA_ZPL_GID(zfsvfs), NULL, &new_gid,
3098                                     sizeof (new_gid));
3099                                 attrzp->z_gid = new_gid;
3100                         }
3101                 }
3102                 if (!(mask & AT_MODE)) {
3103                         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs),
3104                             NULL, &new_mode, sizeof (new_mode));
3105                         new_mode = zp->z_mode;
3106                 }
3107                 err = zfs_acl_chown_setattr(zp);
3108                 ASSERT(err == 0);
3109                 if (attrzp) {
3110                         err = zfs_acl_chown_setattr(attrzp);
3111                         ASSERT(err == 0);
3112                 }
3113         }

3115         if (mask & AT_MODE) {
3116                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs), NULL,
3117                     &new_mode, sizeof (new_mode));
3118                 zp->z_mode = new_mode;
3119                 ASSERT3U((uintptr_t)aclp, !=, NULL);
3120                 err = zfs_aclset_common(zp, aclp, cr, tx);
3121                 ASSERT0(err);
```



```
3122                 if (zp->z_acl_cached)
3123                         zfs_acl_free(zp->z_acl_cached);
3124                 zp->z_acl_cached = aclp;
3125                 aclp = NULL;
3126         }


3129         if (mask & AT_ATIME) {
3130                 ZFS_TIME_ENCODE(&vap->va_atime, zp->z_atime);
3131                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ATIME(zfsvfs), NULL,
3132                     &zp->z_atime, sizeof (zp->z_atime));
3133         }

3135         if (mask & AT_MTIME) {
3136                 ZFS_TIME_ENCODE(&vap->va_mtime, mtime);
3137                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
3138                     mtime, sizeof (mtime));
3139         }

3141         /* XXX - shouldn't this be done *before* the ATIME/MTIME checks? */
3142         if (mask & AT_SIZE && !(mask & AT_MTIME)) {
3143                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs),
3144                     NULL, mtime, sizeof (mtime));
3145                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3146                     &ctime, sizeof (ctime));
3147                 zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
3148                     B_TRUE);
3149         } else if (mask != 0) {
3150                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3151                     &ctime, sizeof (ctime));
3152                 zfs_tstamp_update_setup(zp, STATE_CHANGED, mtime, ctime,
3153                     B_TRUE);
3154                 if (attrzp) {
3155                         SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3156                             SA_ZPL_CTIME(zfsvfs), NULL,
3157                             &ctime, sizeof (ctime));
3158                         zfs_tstamp_update_setup(attrzp, STATE_CHANGED,
3159                             mtime, ctime, B_TRUE);
3160                 }
3161         }
3162         /*
3163          * Do this after setting timestamps to prevent timestamp
3164          * update from toggling bit
3165          */

3167         if (xoap && (mask & AT_XVATTR)) {

3169                 /*
3170                  * restore trimmed off masks
3171                  * so that return masks can be set for caller.
3172                  */

3174                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_APPENDONLY)) {
3175                         XVA_SET_REQ(xvap, XAT_APPENDONLY);
3176                 }
3177                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_NOUNLINK)) {
3178                         XVA_SET_REQ(xvap, XAT_NOUNLINK);
3179                 }
3180                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_IMMUTABLE)) {
3181                         XVA_SET_REQ(xvap, XAT_IMMUTABLE);
3182                 }
3183                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_NODUMP)) {
3184                         XVA_SET_REQ(xvap, XAT_NODUMP);
3185                 }
3186                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_MODIFIED)) {
3187                         XVA_SET_REQ(xvap, XAT_AV_MODIFIED);
```

```
3188                 }
3189                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_QUARANTINED)) {
3190                         XVA_SET_REQ(xvap, XAT_AV_QUARANTINED);
3191                 }

3193                 if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3194                         ASSERT(vp->v_type == VREG);

3196                 zfs_xvattr_set(zp, xvap, tx);
3197         }

3199         if (fuid_dirtied)
3200                 zfs_fuid_sync(zfsvfs, tx);

3202         if (mask != 0)
3203                 zfs_log_setattr(zilog, tx, TX_SETATTR, zp, vap, mask, fuidp);

3205         mutex_exit(&zp->z_lock);
3206         if (mask & (AT_UID|AT_GID|AT_MODE))
3207                 mutex_exit(&zp->z_acl_lock);

3209         if (attrzp) {
3210                 if (mask & (AT_UID|AT_GID|AT_MODE))
3211                         mutex_exit(&attrzp->z_acl_lock);
3212                 mutex_exit(&attrzp->z_lock);
3213         }
3214 out:
3215         if (err == 0 && attrzp) {
3216                 err2 = sa_bulk_update(attrzp->z_sa_hdl, xattr_bulk,
3217                     xattr_count, tx);
3218                 ASSERT(err2 == 0);
3219         }

3221         if (attrzp)
3222                 VN_RELE(ZTOV(attrzp));

3224         if (aclp)
3225                 zfs_acl_free(aclp);

3227         if (fuidp) {
3228                 zfs_fuid_info_free(fuidp);
3229                 fuidp = NULL;
3230         }

3232         if (err) {
3233                 dmu_tx_abort(tx);
3234                 if (err == ERESTART)
3235                         goto top;
3236         } else {
3237                 err2 = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
3238                 dmu_tx_commit(tx);
3239         }

3241 out2:
3242         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3243                 zil_commit(zilog, 0);

3245         ZFS_EXIT(zfsvfs);
3246         return (err);
3247 }
```

*_____unchanged_portion_omitted_*

```
4072 /*
4073  * Push a page out to disk, klustering if possible.
4074  *
4075  *      IN:     vp      - file to push page to.
```



```
4076  *              pp      - page to push.
4077  *              flags   - additional flags.
4078  *              cr      - credentials of caller.
4079  *
4080  *      OUT:    offp    - start of range pushed.
4081  *              lenp    - len of range pushed.
4082  *
4083  *      RETURN: 0 on success, error code on failure.
4084  *
4085  * NOTE: callers must have locked the page to be pushed.  On
4086  * exit, the page (and all other pages in the kluster) must be
4087  * unlocked.
4088  */
4089 /* ARGSUSED */
4090 static int
4091 zfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp,
4092                 size_t *lenp, int flags, cred_t *cr)
4093 {
4094         znode_t         *zp = VTOZ(vp);
4095         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4096         dmu_tx_t        *tx;
4097         u_offset_t      off, koff;
4098         size_t          len, klen;
4099         int             err;

4101         off = pp->p_offset;
4102         len = PAGESIZE;
4103         /*
4104          * If our blocksize is bigger than the page size, try to kluster
4105          * multiple pages so that we write a full block (thus avoiding
4106          * a read-modify-write).
4107          */
4108         if (off < zp->z_size && zp->z_blksz > PAGESIZE) {
4109                 klen = P2ROUNDUP((ulong_t)zp->z_blksz, PAGESIZE);
4110                 koff = ISP2(klen) ? P2ALIGN(off, (u_offset_t)klen) : 0;
4111                 ASSERT(koff <= zp->z_size);
4112                 if (koff + klen > zp->z_size)
4113                         klen = P2ROUNDUP(zp->z_size - koff, (uint64_t)PAGESIZE);
4114                 pp = pvn_write_kluster(vp, pp, &off, &len, koff, klen, flags);
4115         }
4116         ASSERT3U(btop(len), ==, btopr(len));

4118         /*
4119          * Can't push pages past end-of-file.
4120          */
4121         if (off >= zp->z_size) {
4122                 /* ignore all pages */
4123                 err = 0;
4124                 goto out;
4125         } else if (off + len > zp->z_size) {
4126                 int npages = btopr(zp->z_size - off);
4127                 page_t *trunc;

4129                 page_list_break(&pp, &trunc, npages);
4130                 /* ignore pages past end of file */
4131                 if (trunc)
4132                         pvn_write_done(trunc, flags);
4133                 len = zp->z_size - off;
4134         }

4136         if (zfs_owner_overquota(zfsvfs, zp, B_FALSE) ||
4137             zfs_owner_overquota(zfsvfs, zp, B_TRUE)) {
4138                 err = SET_ERROR(EDQUOT);
4139                 goto out;
4140         }
4143 top:
```

```
4141         tx = dmu_tx_create(zfsvfs->z_os);
4142         dmu_tx_hold_write(tx, zp->z_id, off, len);

4144         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
4145         zfs_sa_upgrade_txholds(tx, zp);
4146         err = dmu_tx_assign(tx, TXG_WAIT);
4149         err = dmu_tx_assign(tx, TXG_NOWAIT);
4147         if (err != 0) {
4151                 if (err == ERESTART) {
4152                         dmu_tx_wait(tx);
4148                 dmu_tx_abort(tx);
4154                         goto top;
4155                 }
4156                 dmu_tx_abort(tx);
4149                 goto out;
4150         }

4152         if (zp->z_blksz <= PAGESIZE) {
4153                 caddr_t va = zfs_map_page(pp, S_READ);
4154                 ASSERT3U(len, <=, PAGESIZE);
4155                 dmu_write(zfsvfs->z_os, zp->z_id, off, len, va, tx);
4156                 zfs_unmap_page(pp, va);
4157         } else {
4158                 err = dmu_write_pages(zfsvfs->z_os, zp->z_id, off, len, pp, tx);
4159         }

4161         if (err == 0) {
4162                 uint64_t mtime[2], ctime[2];
4163                 sa_bulk_attr_t bulk[3];
4164                 int count = 0;

4166                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
4167                     &mtime, 16);
4168                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
4169                     &ctime, 16);
4170                 SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
4171                     &zp->z_pflags, 8);
4172                 zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
4173                     B_TRUE);
4174                 zfs_log_write(zfsvfs->z_log, tx, TX_WRITE, zp, off, len, 0);
4175         }
4176         dmu_tx_commit(tx);

4178 out:
4179         pvn_write_done(pp, (err ? B_ERROR : 0) | flags);
4180         if (offp)
4181                 *offp = off;
4182         if (lenp)
4183                 *lenp = len;

4185         return (err);
4186 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   53213 Fri Nov 22 15:15:18 2013
new/usr/src/uts/common/fs/zfs/zfs_znode.c
4347 ZPL can use dmu_tx_assign(TXG_WAIT)
Reviewed by: George Wilson <george.wilson@delphix.com>
Reviewed by: Adam Leventhal <ahl@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1443 /*
1444  * Increase the file length
1445  *
1446  *      IN:     zp      - znode of file to free data in.
1447  *              end     - new end-of-file
1448  *
1449  *      RETURN: 0 on success, error code on failure
1450  */
1451 static int
1452 zfs_extend(znode_t *zp, uint64_t end)
1453 {
1454         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1455         dmu_tx_t *tx;
1456         rl_t *rl;
1457         uint64_t newblksz;
1458         int error;

1460         /*
1461          * We will change zp_size, lock the whole file.
1462          */
1463         rl = zfs_range_lock(zp, 0, UINT64_MAX, RL_WRITER);

1465         /*
1466          * Nothing to do if file already at desired length.
1467          */
1468         if (end <= zp->z_size) {
1469                 zfs_range_unlock(rl);
1470                 return (0);
1471         }
1472 top:
1472         tx = dmu_tx_create(zfsvfs->z_os);
1473         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1474         zfs_sa_upgrade_txholds(tx, zp);
1475         if (end > zp->z_blksz &&
1476             (!ISP2(zp->z_blksz) || zp->z_blksz < zfsvfs->z_max_blksz)) {
1477                 /*
1478                  * We are growing the file past the current block size.
1479                  */
1480                 if (zp->z_blksz > zp->z_zfsvfs->z_max_blksz) {
1481                         ASSERT(!ISP2(zp->z_blksz));
1482                         newblksz = MIN(end, SPA_MAXBLOCKSIZE);
1483                 } else {
1484                         newblksz = MIN(end, zp->z_zfsvfs->z_max_blksz);
1485                 }
1486                 dmu_tx_hold_write(tx, zp->z_id, 0, newblksz);
1487         } else {
1488                 newblksz = 0;
1489         }

1491         error = dmu_tx_assign(tx, TXG_WAIT);
1492         error = dmu_tx_assign(tx, TXG_NOWAIT);
1492         if (error) {
1494                 if (error == ERESTART) {
1495                         dmu_tx_wait(tx);
1493                 dmu_tx_abort(tx);
1497                         goto top;
1498                 }
```



```
1499                 dmu_tx_abort(tx);
1494                 zfs_range_unlock(rl);
1495                 return (error);
1496         }

1498         if (newblksz)
1499                 zfs_grow_blocksize(zp, newblksz, tx);

1501         zp->z_size = end;

1503         VERIFY(0 == sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zp->z_zfsvfs),
1504             &zp->z_size, sizeof (zp->z_size), tx));

1506         zfs_range_unlock(rl);

1508         dmu_tx_commit(tx);

1510         return (0);
1511 }
______unchanged_portion_omitted_

1552 /*
1553  * Truncate a file
1554  *
1555  *      IN:     zp      - znode of file to free data in.
1556  *              end     - new end-of-file.
1557  *
1558  *      RETURN: 0 on success, error code on failure
1559  */
1560 static int
1561 zfs_trunc(znode_t *zp, uint64_t end)
1562 {
1563         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1564         vnode_t *vp = ZTOV(zp);
1565         dmu_tx_t *tx;
1566         rl_t *rl;
1567         int error;
1568         sa_bulk_attr_t bulk[2];
1569         int count = 0;

1571         /*
1572          * We will change zp_size, lock the whole file.
1573          */
1574         rl = zfs_range_lock(zp, 0, UINT64_MAX, RL_WRITER);

1576         /*
1577          * Nothing to do if file already at desired length.
1578          */
1579         if (end >= zp->z_size) {
1580                 zfs_range_unlock(rl);
1581                 return (0);
1582         }

1584         error = dmu_free_long_range(zfsvfs->z_os, zp->z_id, end,  -1);
1585         if (error) {
1586                 zfs_range_unlock(rl);
1587                 return (error);
1588         }
1595 top:
1589         tx = dmu_tx_create(zfsvfs->z_os);
1590         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1591         zfs_sa_upgrade_txholds(tx, zp);
1592         error = dmu_tx_assign(tx, TXG_WAIT);
1599         error = dmu_tx_assign(tx, TXG_NOWAIT);
1593         if (error) {
1601                 if (error == ERESTART) {
```

```
1602                        dmu_tx_wait(tx);
1594                dmu_tx_abort(tx);
1604                        goto top;
1605                }
1606                dmu_tx_abort(tx);
1595                zfs_range_unlock(rl);
1596                return (error);
1597        }

1599        zp->z_size = end;
1600        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_SIZE(zfsvfs),
1601            NULL, &zp->z_size, sizeof (zp->z_size));

1603        if (end == 0) {
1604                zp->z_pflags &= ~ZFS_SPARSE;
1605                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs),
1606                    NULL, &zp->z_pflags, 8);
1607        }
1608        VERIFY(sa_bulk_update(zp->z_sa_hdl, bulk, count, tx) == 0);

1610        dmu_tx_commit(tx);

1612        /*
1613         * Clear any mapped pages in the truncated region.  This has to
1614         * happen outside of the transaction to avoid the possibility of
1615         * a deadlock with someone trying to push a page that we are
1616         * about to invalidate.
1617         */
1618        if (vn_has_cached_data(vp)) {
1619                page_t *pp;
1620                uint64_t start = end & PAGEMASK;
1621                int poff = end & PAGEOFFSET;

1623                if (poff != 0 && (pp = page_lookup(vp, start, SE_SHARED))) {
1624                        /*
1625                         * We need to zero a partial page.
1626                         */
1627                        pagezero(pp, poff, PAGESIZE - poff);
1628                        start += PAGESIZE;
1629                        page_unlock(pp);
1630                }
1631                error = pvn_vplist_dirty(vp, start, zfs_no_putpage,
1632                    B_INVAL | B_TRUNC, NULL);
1633                ASSERT(error == 0);
1634        }

1636        zfs_range_unlock(rl);

1638        return (0);
1639 }

1641 /*
1642  * Free space in a file
1643  *
1644  *      IN:     zp      - znode of file to free data in.
1645  *              off     - start of range
1646  *              len     - end of range (0 => EOF)
1647  *              flag    - current file open mode flags.
1648  *              log     - TRUE if this action should be logged
1649  *
1650  *      RETURN: 0 on success, error code on failure
1651  */
1652 int
1653 zfs_freesp(znode_t *zp, uint64_t off, uint64_t len, int flag, boolean_t log)
1654 {
1655        vnode_t *vp = ZTOV(zp);
```



```
1656        dmu_tx_t *tx;
1657        zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1658        zilog_t *zilog = zfsvfs->z_log;
1659        uint64_t mode;
1660        uint64_t mtime[2], ctime[2];
1661        sa_bulk_attr_t bulk[3];
1662        int count = 0;
1663        int error;

1665        if ((error = sa_lookup(zp->z_sa_hdl, SA_ZPL_MODE(zfsvfs), &mode,
1666            sizeof (mode))) != 0)
1667                return (error);

1669        if (off > zp->z_size) {
1670                error =  zfs_extend(zp, off+len);
1671                if (error == 0 && log)
1672                        goto log;
1673                else
1674                        return (error);
1675        }

1677        /*
1678         * Check for any locks in the region to be freed.
1679         */

1681        if (MANDLOCK(vp, (mode_t)mode)) {
1682                uint64_t length = (len ? len : zp->z_size - off);
1683                if (error = chklock(vp, FWRITE, off, length, flag, NULL))
1684                        return (error);
1685        }

1687        if (len == 0) {
1688                error = zfs_trunc(zp, off);
1689        } else {
1690                if ((error = zfs_free_range(zp, off, len)) == 0 &&
1691                    off + len > zp->z_size)
1692                        error = zfs_extend(zp, off+len);
1693        }
1694        if (error || !log)
1695                return (error);
1696 log:
1697        tx = dmu_tx_create(zfsvfs->z_os);
1698        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1699        zfs_sa_upgrade_txholds(tx, zp);
1700        error = dmu_tx_assign(tx, TXG_WAIT);
1712        error = dmu_tx_assign(tx, TXG_NOWAIT);
1701        if (error) {
1714                if (error == ERESTART) {
1715                        dmu_tx_wait(tx);
1702                dmu_tx_abort(tx);
1717                        goto log;
1718                }
1719                dmu_tx_abort(tx);
1703                return (error);
1704        }

1706        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, mtime, 16);
1707        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, ctime, 16);
1708        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs),
1709            NULL, &zp->z_pflags, 8);
1710        zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime, B_TRUE);
1711        error = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
1712        ASSERT(error == 0);

1714        zfs_log_truncate(zilog, tx, TX_TRUNCATE, zp, off, len);
```

```
1716         dmu_tx_commit(tx);
1717         return (0);
1718 }
```

*unchanged_portion_omitted_*