****
6709 Tue Oct 28 11:57:18 2014
new/usr/src/common/zfs/zfeature_common.c
Possibility to physically reserve space without writing leaf blocks
****
_____***unchanged_portion_omitted_***

```
157 void
158 zpool_feature_init(void)
159 {
160         zfeature_register(SPA_FEATURE_ASYNC_DESTROY,
161             "com.delphix:async_destroy", "async_destroy",
162             "Destroy filesystems asynchronously.", B_TRUE, B_FALSE,
163             B_FALSE, NULL);

165         zfeature_register(SPA_FEATURE_EMPTY_BPOBJ,
166             "com.delphix:empty_bpobj", "empty_bpobj",
167             "Snapshots use less space.", B_TRUE, B_FALSE,
168             B_FALSE, NULL);

170         zfeature_register(SPA_FEATURE_LZ4_COMPRESS,
171             "org.illumos:lz4_compress", "lz4_compress",
172             "LZ4 compression algorithm support.", B_FALSE, B_FALSE,
173             B_TRUE, NULL);

175         zfeature_register(SPA_FEATURE_MULTI_VDEV_CRASH_DUMP,
176             "com.joyent:multi_vdev_crash_dump", "multi_vdev_crash_dump",
177             "Crash dumps to multiple vdev pools.", B_FALSE, B_FALSE,
178             B_FALSE, NULL);

180         zfeature_register(SPA_FEATURE_SPACEMAP_HISTOGRAM,
181             "com.delphix:spacemap_histogram", "spacemap_histogram",
182             "Spacemaps maintain space histograms.", B_TRUE, B_FALSE,
183             B_FALSE, NULL);

185         zfeature_register(SPA_FEATURE_ENABLED_TXG,
186             "com.delphix:enabled_txg", "enabled_txg",
187             "Record txg at which a feature is enabled", B_TRUE, B_FALSE,
188             B_FALSE, NULL);

190         static spa_feature_t hole_birth_deps[] = { SPA_FEATURE_ENABLED_TXG,
191             SPA_FEATURE_NONE };
192         zfeature_register(SPA_FEATURE_HOLE_BIRTH,
193             "com.delphix:hole_birth", "hole_birth",
194             "Retain hole birth txg for more precise zfs send",
195             B_FALSE, B_TRUE, B_TRUE, hole_birth_deps);

197         zfeature_register(SPA_FEATURE_EXTENSIBLE_DATASET,
198             "com.delphix:extensible_dataset", "extensible_dataset",
199             "Enhanced dataset functionality, used by other features.",
200             B_FALSE, B_FALSE, B_FALSE, NULL);

202         static const spa_feature_t bookmarks_deps[] = {
203                 SPA_FEATURE_EXTENSIBLE_DATASET,
204                 SPA_FEATURE_NONE
205         };
206         zfeature_register(SPA_FEATURE_BOOKMARKS,
207             "com.delphix:bookmarks", "bookmarks",
208             "\"zfs bookmark\" command",
209             B_TRUE, B_FALSE, B_FALSE, bookmarks_deps);

211         static const spa_feature_t filesystem_limits_deps[] = {
212             SPA_FEATURE_EXTENSIBLE_DATASET,
213             SPA_FEATURE_NONE
214         };
215         zfeature_register(SPA_FEATURE_FS_SS_LIMIT,
```



```
216             "com.joyent:filesystem_limits", "filesystem_limits",
217             "Filesystem and snapshot limits.", B_TRUE, B_FALSE, B_FALSE,
218             filesystem_limits_deps);

220         zfeature_register(SPA_FEATURE_EMBEDDED_DATA,
221             "com.delphix:embedded_data", "embedded_data",
222             "Blocks which compress very well use even less space.",
223             B_FALSE, B_TRUE, B_TRUE, NULL);

225         zfeature_register(SPA_FEATURE_SPACE_RESERVATION,
226             "org.nexenta:space_reservation", "space_reservation",
227             "Possibility to physically reserve space on disk", B_FALSE, B_FALSE,
228             B_FALSE, NULL);
229 #endif /* ! codereview */
230 }
```

```
**********************************************************
    2706 Tue Oct 28 11:57:18 2014
new/usr/src/common/zfs/zfeature_common.h
Possibility to physically reserve space without writing leaf blocks
**********************************************************
```

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

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 */

#ifndef _ZFEATURE_COMMON_H
#define _ZFEATURE_COMMON_H

#include <sys/fs/zfs.h>
#include <sys/inttypes.h>
#include <sys/types.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct zfeature_info;

typedef enum spa_feature {
        SPA_FEATURE_NONE = -1,
        SPA_FEATURE_ASYNC_DESTROY,
        SPA_FEATURE_EMPTY_BPOBJ,
        SPA_FEATURE_LZ4_COMPRESS,
        SPA_FEATURE_MULTI_VDEV_CRASH_DUMP,
        SPA_FEATURE_SPACEMAP_HISTOGRAM,
        SPA_FEATURE_ENABLED_TXG,
        SPA_FEATURE_HOLE_BIRTH,
        SPA_FEATURE_EXTENSIBLE_DATASET,
        SPA_FEATURE_EMBEDDED_DATA,
        SPA_FEATURE_BOOKMARKS,
        SPA_FEATURE_FS_SS_LIMIT,
        SPA_FEATURE_SPACE_RESERVATION,
#endif /* ! codereview */
        SPA_FEATURES
} spa_feature_t;

#define SPA_FEATURE_DISABLED    (-1ULL)

typedef struct zfeature_info {
```



```
        spa_feature_t fi_feature;
        const char *fi_uname;   /* User-facing feature name */
        const char *fi_guid;    /* On-disk feature identifier */
        const char *fi_desc;    /* Feature description */
        boolean_t fi_can_readonly; /* Can open pool readonly w/o support? */
        boolean_t fi_mos;       /* Is the feature necessary to read the MOS? */
        /* Activate this feature at the same time it is enabled */
        boolean_t fi_activate_on_enable;
        /* array of dependencies, terminated by SPA_FEATURE_NONE */
        const spa_feature_t *fi_depends;
} zfeature_info_t;

typedef int (zfeature_func_t)(zfeature_info_t *, void *);

#define ZFS_FEATURE_DEBUG

extern zfeature_info_t spa_feature_table[SPA_FEATURES];

extern boolean_t zfeature_is_valid_guid(const char *);

extern boolean_t zfeature_is_supported(const char *);
extern int zfeature_lookup_name(const char *, spa_feature_t *);
extern boolean_t zfeature_depends_on(spa_feature_t, spa_feature_t);

extern void zpool_feature_init(void);

#ifdef  __cplusplus
}
#endif

#endif  /* _ZFEATURE_COMMON_H */
```

```
**********************************************************
   79384 Tue Oct 28 11:57:18 2014
new/usr/src/uts/common/fs/zfs/dbuf.c
Possibility to physically reserve space without writing leaf blocks
**********************************************************
______unchanged_portion_omitted_

1023 dbuf_dirty_record_t *
1024 dbuf_zero_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1025 {
1026         ASSERT(db->db_objset != NULL);

1028         return (dbuf_dirty(db, tx, B_TRUE));
1029 }

1031 dbuf_dirty_record_t *
1032 dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx, boolean_t zero_write)
1024 dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx)
1033 {
1034         dnode_t *dn;
1035         objset_t *os;
1036         dbuf_dirty_record_t **drp, *dr;
1037         int drop_struct_lock = FALSE;
1038         boolean_t do_free_accounting = B_FALSE;
1039         int txgoff = tx->tx_txg & TXG_MASK;

1041         ASSERT(tx->tx_txg != 0);
1042         ASSERT(!refcount_is_zero(&db->db_holds));
1043         DMU_TX_DIRTY_BUF(tx, db);

1045         DB_DNODE_ENTER(db);
1046         dn = DB_DNODE(db);
1047         /*
1048          * Shouldn't dirty a regular buffer in syncing context.  Private
1049          * objects may be dirtied in syncing context, but only if they
1050          * were already pre-dirtied in open context.
1051          */
1052         ASSERT(!dmu_tx_is_syncing(tx) ||
1053             BP_IS_HOLE(dn->dn_objset->os_rootbp) ||
1054             DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1055             dn->dn_objset->os_dsl_dataset == NULL);
1056         /*
1057          * We make this assert for private objects as well, but after we
1058          * check if we're already dirty.  They are allowed to re-dirty
1059          * in syncing context.
1060          */
1061         ASSERT(dn->dn_object == DMU_META_DNODE_OBJECT ||
1062             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1063             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1065         mutex_enter(&db->db_mtx);
1066         /*
1067          * XXX make this true for indirects too?  The problem is that
1068          * transactions created with dmu_tx_create_assigned() from
1069          * syncing context don't bother holding ahead.
1070          */
1071         ASSERT(db->db_level != 0 ||
1072             db->db_state == DB_CACHED || db->db_state == DB_FILL ||
1073             db->db_state == DB_NOFILL);

1075         mutex_enter(&dn->dn_mtx);
1076         /*
1077          * Don't set dirtyctx to SYNC if we're just modifying this as we
1078          * initialize the objset.
1079          */
1080         if (dn->dn_dirtyctx == DN_UNDIRTIED &&
```



```
1081             !BP_IS_HOLE(dn->dn_objset->os_rootbp)) {
1082                 dn->dn_dirtyctx =
1083                     (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN);
1084                 ASSERT(dn->dn_dirtyctx_firstset == NULL);
1085                 dn->dn_dirtyctx_firstset = kmem_alloc(1, KM_SLEEP);
1086         }
1087         mutex_exit(&dn->dn_mtx);

1089         if (db->db_blkid == DMU_SPILL_BLKID)
1090                 dn->dn_have_spill = B_TRUE;

1092         /*
1093          * If this buffer is already dirty, we're done.
1094          */
1095         drp = &db->db_last_dirty;
1096         ASSERT(*drp == NULL || (*drp)->dr_txg <= tx->tx_txg ||
1097             db->db.db_object == DMU_META_DNODE_OBJECT);
1098         while ((dr = *drp) != NULL && dr->dr_txg > tx->tx_txg)
1099                 drp = &dr->dr_next;
1100         if (dr && dr->dr_txg == tx->tx_txg) {
1101                 DB_DNODE_EXIT(db);

1103                 if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID) {
1104                         /*
1105                          * If this buffer has already been written out,
1106                          * we now need to reset its state.
1107                          */
1108                         dbuf_unoverride(dr);
1109                         if (db->db.db_object != DMU_META_DNODE_OBJECT &&
1110                             db->db_state != DB_NOFILL)
1111                                 arc_buf_thaw(db->db_buf);
1112                 }
1113                 mutex_exit(&db->db_mtx);
1114                 return (dr);
1115         }

1117         /*
1118          * Only valid if not already dirty.
1119          */
1120         ASSERT(dn->dn_object == 0 ||
1121             dn->dn_dirtyctx == DN_UNDIRTIED || dn->dn_dirtyctx ==
1122             (dmu_tx_is_syncing(tx) ? DN_DIRTY_SYNC : DN_DIRTY_OPEN));

1124         ASSERT3U(dn->dn_nlevels, >, db->db_level);
1125         ASSERT((dn->dn_phys->dn_nlevels == 0 && db->db_level == 0) ||
1126             dn->dn_phys->dn_nlevels > db->db_level ||
1127             dn->dn_next_nlevels[txgoff] > db->db_level ||
1128             dn->dn_next_nlevels[(tx->tx_txg-1) & TXG_MASK] > db->db_level ||
1129             dn->dn_next_nlevels[(tx->tx_txg-2) & TXG_MASK] > db->db_level);

1131         /*
1132          * We should only be dirtying in syncing context if it's the
1133          * mos or we're initializing the os or it's a special object.
1134          * However, we are allowed to dirty in syncing context provided
1135          * we already dirtied it in open context.  Hence we must make
1136          * this assertion only if we're not already dirty.
1137          */
1138         os = dn->dn_objset;
1139         ASSERT(!dmu_tx_is_syncing(tx) || DMU_OBJECT_IS_SPECIAL(dn->dn_object) ||
1140             os->os_dsl_dataset == NULL || BP_IS_HOLE(os->os_rootbp));
1141         ASSERT(db->db.db_size != 0);

1143         dprintf_dbuf(db, "size=%llx\n", (u_longlong_t)db->db.db_size);

1145         if (db->db_blkid != DMU_BONUS_BLKID) {
1146                 /*
```

```
1147                 * Update the accounting.
1148                 * Note: we delay "free accounting" until after we drop
1149                 * the db_mtx.  This keeps us from grabbing other locks
1150                 * (and possibly deadlocking) in bp_get_dsize() while
1151                 * also holding the db_mtx.
1152                 */
1153                dnode_willuse_space(dn, db->db.db_size, tx);
1154                do_free_accounting = dbuf_block_freeable(db);
1155        }

1157        /*
1158         * If this buffer is dirty in an old transaction group we need
1159         * to make a copy of it so that the changes we make in this
1160         * transaction group won't leak out when we sync the older txg.
1161         */
1162        dr = kmem_zalloc(sizeof (dbuf_dirty_record_t), KM_SLEEP);
1163        dr->dr_zero_write = zero_write;
1164 #endif /* ! codereview */
1165        if (db->db_level == 0) {
1166                void *data_old = db->db_buf;

1168                if (db->db_state != DB_NOFILL) {
1169                        if (db->db_blkid == DMU_BONUS_BLKID) {
1170                                dbuf_fix_old_data(db, tx->tx_txg);
1171                                data_old = db->db.db_data;
1172                        } else if (db->db.db_object != DMU_META_DNODE_OBJECT) {
1173                                /*
1174                                 * Release the data buffer from the cache so
1175                                 * that we can modify it without impacting
1176                                 * possible other users of this cached data
1177                                 * block.  Note that indirect blocks and
1178                                 * private objects are not released until the
1179                                 * syncing state (since they are only modified
1180                                 * then).
1181                                 */
1182                                arc_release(db->db_buf, db);
1183                                dbuf_fix_old_data(db, tx->tx_txg);
1184                                data_old = db->db_buf;
1185                        }
1186                        ASSERT(data_old != NULL);
1187                }
1188                dr->dt.dl.dr_data = data_old;
1189        } else {
1190                mutex_init(&dr->dt.di.dr_mtx, NULL, MUTEX_DEFAULT, NULL);
1191                list_create(&dr->dt.di.dr_children,
1192                    sizeof (dbuf_dirty_record_t),
1193                    offsetof(dbuf_dirty_record_t, dr_dirty_node));
1194        }
1195        if (db->db_blkid != DMU_BONUS_BLKID && os->os_dsl_dataset != NULL)
1196                dr->dr_accounted = db->db.db_size;
1197        dr->dr_dbuf = db;
1198        dr->dr_txg = tx->tx_txg;
1199        dr->dr_next = *drp;
1200        *drp = dr;

1202        /*
1203         * We could have been freed_in_flight between the dbuf_noread
1204         * and dbuf_dirty.  We win, as though the dbuf_noread() had
1205         * happened after the free.
1206         */
1207        if (db->db_level == 0 && db->db_blkid != DMU_BONUS_BLKID &&
1208            db->db_blkid != DMU_SPILL_BLKID) {
1209                mutex_enter(&dn->dn_mtx);
1210                if (dn->dn_free_ranges[txgoff] != NULL) {
1211                        range_tree_clear(dn->dn_free_ranges[txgoff],
1212                            db->db_blkid, 1);
```



```
1213                }
1214                mutex_exit(&dn->dn_mtx);
1215                db->db_freed_in_flight = FALSE;
1216        }

1218        /*
1219         * This buffer is now part of this txg
1220         */
1221        dbuf_add_ref(db, (void *)(uintptr_t)tx->tx_txg);
1222        db->db_dirtycnt += 1;
1223        ASSERT3U(db->db_dirtycnt, <=, 3);

1225        mutex_exit(&db->db_mtx);

1227        if (db->db_blkid == DMU_BONUS_BLKID ||
1228            db->db_blkid == DMU_SPILL_BLKID) {
1229                mutex_enter(&dn->dn_mtx);
1230                ASSERT(!list_link_active(&dr->dr_dirty_node));
1231                list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1232                mutex_exit(&dn->dn_mtx);
1233                dnode_setdirty(dn, tx);
1234                DB_DNODE_EXIT(db);
1235                return (dr);
1236        } else if (do_free_accounting) {
1237                blkptr_t *bp = db->db_blkptr;
1238                int64_t willfree = (bp && !BP_IS_HOLE(bp)) ?
1239                    bp_get_dsize(os->os_spa, bp) : db->db.db_size;
1240                /*
1241                 * This is only a guess -- if the dbuf is dirty
1242                 * in a previous txg, we don't know how much
1243                 * space it will use on disk yet.  We should
1244                 * really have the struct_rwlock to access
1245                 * db_blkptr, but since this is just a guess,
1246                 * it's OK if we get an odd answer.
1247                 */
1248                ddt_prefetch(os->os_spa, bp);
1249                dnode_willuse_space(dn, -willfree, tx);
1250        }

1252        if (!RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
1253                rw_enter(&dn->dn_struct_rwlock, RW_READER);
1254                drop_struct_lock = TRUE;
1255        }

1257        if (db->db_level == 0) {
1258                dnode_new_blkid(dn, db->db_blkid, tx, drop_struct_lock);
1259                ASSERT(dn->dn_maxblkid >= db->db_blkid);
1260        }

1262        if (db->db_level+1 < dn->dn_nlevels) {
1263                dmu_buf_impl_t *parent = db->db_parent;
1264                dbuf_dirty_record_t *di;
1265                int parent_held = FALSE;

1267                if (db->db_parent == NULL || db->db_parent == dn->dn_dbuf) {
1268                        int epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;

1270                        parent = dbuf_hold_level(dn, db->db_level+1,
1271                            db->db_blkid >> epbs, FTAG);
1272                        ASSERT(parent != NULL);
1273                        parent_held = TRUE;
1274                }
1275                if (drop_struct_lock)
1276                        rw_exit(&dn->dn_struct_rwlock);
1277                ASSERT3U(db->db_level+1, ==, parent->db_level);
1278                di = dbuf_dirty(parent, tx, B_FALSE);
```

```
1155                 di = dbuf_dirty(parent, tx);
1279                 if (parent_held)
1280                         dbuf_rele(parent, FTAG);

1282                 mutex_enter(&db->db_mtx);
1283                 /*
1284                  * Since we've dropped the mutex, it's possible that
1285                  * dbuf_undirty() might have changed this out from under us.
1286                  */
1287                 if (db->db_last_dirty == dr ||
1288                     dn->dn_object == DMU_META_DNODE_OBJECT) {
1289                         mutex_enter(&di->dt.di.dr_mtx);
1290                         ASSERT3U(di->dr_txg, ==, tx->tx_txg);
1291                         ASSERT(!list_link_active(&dr->dr_dirty_node));
1292                         list_insert_tail(&di->dt.di.dr_children, dr);
1293                         mutex_exit(&di->dt.di.dr_mtx);
1294                         dr->dr_parent = di;
1295                 }
1296                 mutex_exit(&db->db_mtx);
1297         } else {
1298                 ASSERT(db->db_level+1 == dn->dn_nlevels);
1299                 ASSERT(db->db_blkid < dn->dn_nblkptr);
1300                 ASSERT(db->db_parent == NULL || db->db_parent == dn->dn_dbuf);
1301                 mutex_enter(&dn->dn_mtx);
1302                 ASSERT(!list_link_active(&dr->dr_dirty_node));
1303                 list_insert_tail(&dn->dn_dirty_records[txgoff], dr);
1304                 mutex_exit(&dn->dn_mtx);
1305                 if (drop_struct_lock)
1306                         rw_exit(&dn->dn_struct_rwlock);
1307         }

1309         dnode_setdirty(dn, tx);
1310         DB_DNODE_EXIT(db);
1311         return (dr);
1312 }
______unchanged_portion_omitted_

1407 void
1408 dmu_buf_will_dirty(dmu_buf_t *db_fake, dmu_tx_t *tx)
1409 {
1410         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
1411         int rf = DB_RF_MUST_SUCCEED | DB_RF_NOPREFETCH;

1413         ASSERT(tx->tx_txg != 0);
1414         ASSERT(!refcount_is_zero(&db->db_holds));

1416         DB_DNODE_ENTER(db);
1417         if (RW_WRITE_HELD(&DB_DNODE(db)->dn_struct_rwlock))
1418                 rf |= DB_RF_HAVESTRUCT;
1419         DB_DNODE_EXIT(db);
1420         (void) dbuf_read(db, NULL, rf);
1421         (void) dbuf_dirty(db, tx, B_FALSE);
1298         (void) dbuf_dirty(db, tx);
1422 }
______unchanged_portion_omitted_

1434 void
1435 dmu_buf_will_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1436 {
1437         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1439         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1440         ASSERT(tx->tx_txg != 0);
1441         ASSERT(db->db_level == 0);
1442         ASSERT(!refcount_is_zero(&db->db_holds));
```



```
1444         ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT ||
1445             dmu_tx_private_ok(tx));

1447         dbuf_noread(db);
1448         (void) dbuf_dirty(db, tx, B_FALSE);
1449 }


1452 void
1453 dmu_buf_will_zero_fill(dmu_buf_t *db_fake, dmu_tx_t *tx)
1454 {
1455         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1457         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1458         ASSERT(tx->tx_txg != 0);
1459         ASSERT(db->db_level == 0);
1460         ASSERT(!refcount_is_zero(&db->db_holds));

1462         ASSERT(db->db.db_object != DMU_META_DNODE_OBJECT ||
1463             dmu_tx_private_ok(tx));

1465         dbuf_noread(db);
1466         (void) dbuf_zero_dirty(db, tx);
1325         (void) dbuf_dirty(db, tx);
1467 }
______unchanged_portion_omitted_

1523 /*
1524  * Directly assign a provided arc buf to a given dbuf if it's not referenced
1525  * by anybody except our caller. Otherwise copy arcbuf's contents to dbuf.
1526  */
1527 void
1528 dbuf_assign_arcbuf(dmu_buf_impl_t *db, arc_buf_t *buf, dmu_tx_t *tx)
1529 {
1530         ASSERT(!refcount_is_zero(&db->db_holds));
1531         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
1532         ASSERT(db->db_level == 0);
1533         ASSERT(DBUF_GET_BUFC_TYPE(db) == ARC_BUFC_DATA);
1534         ASSERT(buf != NULL);
1535         ASSERT(arc_buf_size(buf) == db->db.db_size);
1536         ASSERT(tx->tx_txg != 0);

1538         arc_return_buf(buf, db);
1539         ASSERT(arc_released(buf));

1541         mutex_enter(&db->db_mtx);

1543         while (db->db_state == DB_READ || db->db_state == DB_FILL)
1544                 cv_wait(&db->db_changed, &db->db_mtx);

1546         ASSERT(db->db_state == DB_CACHED || db->db_state == DB_UNCACHED);

1548         if (db->db_state == DB_CACHED &&
1549             refcount_count(&db->db_holds) - 1 > db->db_dirtycnt) {
1550                 mutex_exit(&db->db_mtx);
1551                 (void) dbuf_dirty(db, tx, B_FALSE);
1410                 (void) dbuf_dirty(db, tx);
1552                 bcopy(buf->b_data, db->db.db_data, db->db.db_size);
1553                 VERIFY(arc_buf_remove_ref(buf, db));
1554                 xuio_stat_wbuf_copied();
1555                 return;
1556         }

1558         xuio_stat_wbuf_nocopy();
1559         if (db->db_state == DB_CACHED) {
1560                 dbuf_dirty_record_t *dr = db->db_last_dirty;
```

```
1562                 ASSERT(db->db_buf != NULL);
1563                 if (dr != NULL && dr->dr_txg == tx->tx_txg) {
1564                         ASSERT(dr->dt.dl.dr_data == db->db_buf);
1565                         if (!arc_released(db->db_buf)) {
1566                                 ASSERT(dr->dt.dl.dr_override_state ==
1567                                     DR_OVERRIDDEN);
1568                                 arc_release(db->db_buf, db);
1569                         }
1570                         dr->dt.dl.dr_data = buf;
1571                         VERIFY(arc_buf_remove_ref(db->db_buf, db));
1572                 } else if (dr == NULL || dr->dt.dl.dr_data != db->db_buf) {
1573                         arc_release(db->db_buf, db);
1574                         VERIFY(arc_buf_remove_ref(db->db_buf, db));
1575                 }
1576                 db->db_buf = NULL;
1577         }
1578         ASSERT(db->db_buf == NULL);
1579         dbuf_set_data(db, buf);
1580         db->db_state = DB_FILL;
1581         mutex_exit(&db->db_mtx);
1582         (void) dbuf_dirty(db, tx, B_FALSE);
1441         (void) dbuf_dirty(db, tx);
1583         dmu_buf_fill_done(&db->db, tx);
1584 }
_____unchanged_portion_omitted_

2779 /* Issue I/O to commit a dirty buffer to disk. */
2780 static void
2781 dbuf_write(dbuf_dirty_record_t *dr, arc_buf_t *data, dmu_tx_t *tx)
2782 {
2783         dmu_buf_impl_t *db = dr->dr_dbuf;
2784         dnode_t *dn;
2785         objset_t *os;
2786         dmu_buf_impl_t *parent = db->db_parent;
2787         uint64_t txg = tx->tx_txg;
2788         zbookmark_phys_t zb;
2789         zio_prop_t zp;
2790         zio_t *zio;
2791         int wp_flag = 0;

2793         DB_DNODE_ENTER(db);
2794         dn = DB_DNODE(db);
2795         os = dn->dn_objset;

2797         if (db->db_state != DB_NOFILL) {
2798                 if (db->db_level > 0 || dn->dn_type == DMU_OT_DNODE) {
2799                         /*
2800                          * Private object buffers are released here rather
2801                          * than in dbuf_dirty() since they are only modified
2802                          * in the syncing context and we don't want the
2803                          * overhead of making multiple copies of the data.
2804                          */
2805                         if (BP_IS_HOLE(db->db_blkptr)) {
2806                                 arc_buf_thaw(data);
2807                         } else {
2808                                 dbuf_release_bp(db);
2809                         }
2810                 }
2811         }

2813         if (parent != dn->dn_dbuf) {
2814                 /* Our parent is an indirect block. */
2815                 /* We have a dirty parent that has been scheduled for write. */
2816                 ASSERT(parent && parent->db_data_pending);
2817                 /* Our parent's buffer is one level closer to the dnode. */
```



```
2818                 ASSERT(db->db_level == parent->db_level-1);
2819                 /*
2820                  * We're about to modify our parent's db_data by modifying
2821                  * our block pointer, so the parent must be released.
2822                  */
2823                 ASSERT(arc_released(parent->db_buf));
2824                 zio = parent->db_data_pending->dr_zio;
2825         } else {
2826                 /* Our parent is the dnode itself. */
2827                 ASSERT((db->db_level == dn->dn_phys->dn_nlevels-1 &&
2828                     db->db_blkid != DMU_SPILL_BLKID) ||
2829                     (db->db_blkid == DMU_SPILL_BLKID && db->db_level == 0));
2830                 if (db->db_blkid != DMU_SPILL_BLKID)
2831                         ASSERT3P(db->db_blkptr, ==,
2832                             &dn->dn_phys->dn_blkptr[db->db_blkid]);
2833                 zio = dn->dn_zio;
2834         }

2836         ASSERT(db->db_level == 0 || data == db->db_buf);
2837         ASSERT3U(db->db_blkptr->blk_birth, <=, txg);
2838         ASSERT(zio);

2840         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
2841             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
2842             db->db.db_object, db->db_level, db->db_blkid);

2844         if (db->db_blkid == DMU_SPILL_BLKID)
2845                 wp_flag = WP_SPILL;
2846         wp_flag |= (db->db_state == DB_NOFILL) ? WP_NOFILL : 0;

2848         dmu_write_policy(os, dn, db->db_level, wp_flag, &zp);
2849         DB_DNODE_EXIT(db);

2851         if (dr->dr_zero_write) {
2852                 zp.zp_zero_write = B_TRUE;

2854                 if (!spa_feature_is_active(os->os_spa, SPA_FEATURE_SPACE_RESERVA
2855                 {
2856                         spa_feature_incr(os->os_spa,
2857                             SPA_FEATURE_SPACE_RESERVATION, tx);
2858                 }
2859         }

2861 #endif /* ! codereview */
2862         if (db->db_level == 0 &&
2863             dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
2864                 /*
2865                  * The BP for this block has been provided by open context
2866                  * (by dmu_sync() or dmu_buf_write_embedded()).
2867                  */
2868                 void *contents = (data != NULL) ? data->b_data : NULL;

2870                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
2871                     db->db_blkptr, contents, db->db.db_size, &zp,
2872                     dbuf_write_override_ready, NULL, dbuf_write_override_done,
2873                     dr, ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
2874                 mutex_enter(&db->db_mtx);
2875                 dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
2876                 zio_write_override(dr->dr_zio, &dr->dt.dl.dr_overridden_by,
2877                     dr->dt.dl.dr_copies, dr->dt.dl.dr_nopwrite);
2878                 mutex_exit(&db->db_mtx);
2879         } else if (db->db_state == DB_NOFILL) {
2880                 ASSERT(zp.zp_checksum == ZIO_CHECKSUM_OFF ||
2881                     zp.zp_checksum == ZIO_CHECKSUM_NOPARITY);
2882                 dr->dr_zio = zio_write(zio, os->os_spa, txg,
2883                     db->db_blkptr, NULL, db->db.db_size, &zp,
```

```
2884                     dbuf_write_nofill_ready, NULL, dbuf_write_nofill_done, db,
2885                     ZIO_PRIORITY_ASYNC_WRITE,
2886                     ZIO_FLAG_MUSTSUCCEED | ZIO_FLAG_NODATA, &zb);
2887         } else {
2888                 ASSERT(arc_released(data));
2889                 dr->dr_zio = arc_write(zio, os->os_spa, txg,
2890                     db->db_blkptr, data, DBUF_IS_L2CACHEABLE(db),
2891                     DBUF_IS_L2COMPRESSIBLE(db), &zp, dbuf_write_ready,
2892                     dbuf_write_physdone, dbuf_write_done, db,
2893                     ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);
2894         }
2895 }
```

```
**********************************************************
   49077 Tue Oct 28 11:57:19 2014
new/usr/src/uts/common/fs/zfs/dmu.c
Possibility to physically reserve space without writing leaf blocks
**********************************************************
______unchanged_portion_omitted_
```

```
 812 void
 813 dmu_write_zero(objset_t *os, uint64_t object, uint64_t offset, uint64_t size, dm
 814 {
 815         dmu_buf_t       **dbp;
 816         int             numbufs, i;

 818         VERIFY(0 == dmu_buf_hold_array(os, object, offset, size,
 819             FALSE, FTAG, &numbufs, &dbp));

 821         for (i = 0; i < numbufs; i++) {
 822                 dmu_buf_t *db = dbp[i];

 824                 dmu_buf_will_zero_fill(db, tx);

 826                 memset(db->db_data, 0, db->db_size);

 828                 dmu_buf_fill_done(db, tx);
 829         }

 831         dmu_buf_rele_array(dbp, numbufs, FTAG);
 832 }

 834 void
 835 #endif /* ! codereview */
 836 dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
 837     const void *buf, dmu_tx_t *tx)
 838 {
 839         dmu_buf_t **dbp;
 840         int numbufs, i;

 842         if (size == 0)
 843                 return;

 845         VERIFY(0 == dmu_buf_hold_array(os, object, offset, size,
 846             FALSE, FTAG, &numbufs, &dbp));

 848         for (i = 0; i < numbufs; i++) {
 849                 int tocpy;
 850                 int bufoff;
 851                 dmu_buf_t *db = dbp[i];

 853                 ASSERT(size > 0);

 855                 bufoff = offset - db->db_offset;
 856                 tocpy = (int)MIN(db->db_size - bufoff, size);

 858                 ASSERT(i == 0 || i == numbufs-1 || tocpy == db->db_size);

 860                 if (tocpy == db->db_size)
 861                         dmu_buf_will_fill(db, tx);
 862                 else
 863                         dmu_buf_will_dirty(db, tx);

 865                 bcopy(buf, (char *)db->db_data + bufoff, tocpy);

 867                 if (tocpy == db->db_size)
 868                         dmu_buf_fill_done(db, tx);

 870                 offset += tocpy;
```



```
 871                 size -= tocpy;
 872                 buf = (char *)buf + tocpy;
 873         }
 874         dmu_buf_rele_array(dbp, numbufs, FTAG);
 875 }

 877 void
 878 dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
 879     dmu_tx_t *tx)
 880 {
 881         dmu_buf_t **dbp;
 882         int numbufs, i;

 884         if (size == 0)
 885                 return;

 887         VERIFY(0 == dmu_buf_hold_array(os, object, offset, size,
 888             FALSE, FTAG, &numbufs, &dbp));

 890         for (i = 0; i < numbufs; i++) {
 891                 dmu_buf_t *db = dbp[i];

 893                 dmu_buf_will_not_fill(db, tx);
 894         }
 895         dmu_buf_rele_array(dbp, numbufs, FTAG);
 896 }

 898 void
 899 dmu_write_embedded(objset_t *os, uint64_t object, uint64_t offset,
 900     void *data, uint8_t etype, uint8_t comp, int uncompressed_size,
 901     int compressed_size, int byteorder, dmu_tx_t *tx)
 902 {
 903         dmu_buf_t *db;

 905         ASSERT3U(etype, <, NUM_BP_EMBEDDED_TYPES);
 906         ASSERT3U(comp, <, ZIO_COMPRESS_FUNCTIONS);
 907         VERIFY0(dmu_buf_hold_noread(os, object, offset,
 908             FTAG, &db));

 910         dmu_buf_write_embedded(db,
 911             data, (bp_embedded_type_t)etype, (enum zio_compress)comp,
 912             uncompressed_size, compressed_size, byteorder, tx);

 914         dmu_buf_rele(db, FTAG);
 915 }

 917 /*
 918  * DMU support for xuio
 919  */
 920 kstat_t *xuio_ksp = NULL;

 922 int
 923 dmu_xuio_init(xuio_t *xuio, int nblk)
 924 {
 925         dmu_xuio_t *priv;
 926         uio_t *uio = &xuio->xu_uio;

 928         uio->uio_iovcnt = nblk;
 929         uio->uio_iov = kmem_zalloc(nblk * sizeof (iovec_t), KM_SLEEP);

 931         priv = kmem_zalloc(sizeof (dmu_xuio_t), KM_SLEEP);
 932         priv->cnt = nblk;
 933         priv->bufs = kmem_zalloc(nblk * sizeof (arc_buf_t *), KM_SLEEP);
 934         priv->iovp = uio->uio_iov;
 935         XUIO_XUZC_PRIV(xuio) = priv;
```

```
937         if (XUIO_XUZC_RW(xuio) == UIO_READ)
938                 XUIOSTAT_INCR(xuiostat_onloan_rbuf, nblk);
939         else
940                 XUIOSTAT_INCR(xuiostat_onloan_wbuf, nblk);
941
942         return (0);
943 }
944
945 void
946 dmu_xuio_fini(xuio_t *xuio)
947 {
948         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
949         int nblk = priv->cnt;
950
951         kmem_free(priv->iovp, nblk * sizeof (iovec_t));
952         kmem_free(priv->bufs, nblk * sizeof (arc_buf_t *));
953         kmem_free(priv, sizeof (dmu_xuio_t));
954
955         if (XUIO_XUZC_RW(xuio) == UIO_READ)
956                 XUIOSTAT_INCR(xuiostat_onloan_rbuf, -nblk);
957         else
958                 XUIOSTAT_INCR(xuiostat_onloan_wbuf, -nblk);
959 }
960
961 /*
962  * Initialize iov[priv->next] and priv->bufs[priv->next] with { off, n, abuf }
963  * and increase priv->next by 1.
964  */
965 int
966 dmu_xuio_add(xuio_t *xuio, arc_buf_t *abuf, offset_t off, size_t n)
967 {
968         struct iovec *iov;
969         uio_t *uio = &xuio->xu_uio;
970         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
971         int i = priv->next++;
972
973         ASSERT(i < priv->cnt);
974         ASSERT(off + n <= arc_buf_size(abuf));
975         iov = uio->uio_iov + i;
976         iov->iov_base = (char *)abuf->b_data + off;
977         iov->iov_len = n;
978         priv->bufs[i] = abuf;
979         return (0);
980 }
981
982 int
983 dmu_xuio_cnt(xuio_t *xuio)
984 {
985         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
986         return (priv->cnt);
987 }
988
989 arc_buf_t *
990 dmu_xuio_arcbuf(xuio_t *xuio, int i)
991 {
992         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
993
994         ASSERT(i < priv->cnt);
995         return (priv->bufs[i]);
996 }
997
998 void
999 dmu_xuio_clear(xuio_t *xuio, int i)
1000 {
1001         dmu_xuio_t *priv = XUIO_XUZC_PRIV(xuio);
```



```
1003         ASSERT(i < priv->cnt);
1004         priv->bufs[i] = NULL;
1005 }
1006
1007 static void
1008 xuio_stat_init(void)
1009 {
1010         xuio_ksp = kstat_create("zfs", 0, "xuio_stats", "misc",
1011             KSTAT_TYPE_NAMED, sizeof (xuio_stats) / sizeof (kstat_named_t),
1012             KSTAT_FLAG_VIRTUAL);
1013         if (xuio_ksp != NULL) {
1014                 xuio_ksp->ks_data = &xuio_stats;
1015                 kstat_install(xuio_ksp);
1016         }
1017 }
1018
1019 static void
1020 xuio_stat_fini(void)
1021 {
1022         if (xuio_ksp != NULL) {
1023                 kstat_delete(xuio_ksp);
1024                 xuio_ksp = NULL;
1025         }
1026 }
1027
1028 void
1029 xuio_stat_wbuf_copied()
1030 {
1031         XUIOSTAT_BUMP(xuiostat_wbuf_copied);
1032 }
1033
1034 void
1035 xuio_stat_wbuf_nocopy()
1036 {
1037         XUIOSTAT_BUMP(xuiostat_wbuf_nocopy);
1038 }
1039
1040 #ifdef _KERNEL
1041 static int
1042 dmu_read_uio_dnode(dnode_t *dn, uio_t *uio, uint64_t size)
1043 {
1044         dmu_buf_t **dbp;
1045         int numbufs, i, err;
1046         xuio_t *xuio = NULL;
1047
1048         /*
1049          * NB: we could do this block-at-a-time, but it's nice
1050          * to be reading in parallel.
1051          */
1052         err = dmu_buf_hold_array_by_dnode(dn, uio->uio_loffset, size,
1053             TRUE, FTAG, &numbufs, &dbp, 0);
1054         if (err)
1055                 return (err);
1056
1057         if (uio->uio_extflg == UIO_XUIO)
1058                 xuio = (xuio_t *)uio;
1059
1060         for (i = 0; i < numbufs; i++) {
1061                 int tocpy;
1062                 int bufoff;
1063                 dmu_buf_t *db = dbp[i];
1064
1065                 ASSERT(size > 0);
1066
1067                 bufoff = uio->uio_loffset - db->db_offset;
1068                 tocpy = (int)MIN(db->db_size - bufoff, size);
```

```
1070                 if (xuio) {
1071                         dmu_buf_impl_t *dbi = (dmu_buf_impl_t *)db;
1072                         arc_buf_t *dbuf_abuf = dbi->db_buf;
1073                         arc_buf_t *abuf = dbuf_loan_arcbuf(dbi);
1074                         err = dmu_xuio_add(xuio, abuf, bufoff, tocpy);
1075                         if (!err) {
1076                                 uio->uio_resid -= tocpy;
1077                                 uio->uio_loffset += tocpy;
1078                         }
1079
1080                         if (abuf == dbuf_abuf)
1081                                 XUIOSTAT_BUMP(xuiostat_rbuf_nocopy);
1082                         else
1083                                 XUIOSTAT_BUMP(xuiostat_rbuf_copied);
1084                 } else {
1085                         err = uiomove((char *)db->db_data + bufoff, tocpy,
1086                             UIO_READ, uio);
1087                 }
1088                 if (err)
1089                         break;
1090
1091                 size -= tocpy;
1092         }
1093         dmu_buf_rele_array(dbp, numbufs, FTAG);
1094
1095         return (err);
1096 }
1097
1098 /*
1099  * Read 'size' bytes into the uio buffer.
1100  * From object zdb->db_object.
1101  * Starting at offset uio->uio_loffset.
1102  *
1103  * If the caller already has a dbuf in the target object
1104  * (e.g. its bonus buffer), this routine is faster than dmu_read_uio(),
1105  * because we don't have to find the dnode_t for the object.
1106  */
1107 int
1108 dmu_read_uio_dbuf(dmu_buf_t *zdb, uio_t *uio, uint64_t size)
1109 {
1110         dmu_buf_impl_t *db = (dmu_buf_impl_t *)zdb;
1111         dnode_t *dn;
1112         int err;
1113
1114         if (size == 0)
1115                 return (0);
1116
1117         DB_DNODE_ENTER(db);
1118         dn = DB_DNODE(db);
1119         err = dmu_read_uio_dnode(dn, uio, size);
1120         DB_DNODE_EXIT(db);
1121
1122         return (err);
1123 }
1124
1125 /*
1126  * Read 'size' bytes into the uio buffer.
1127  * From the specified object
1128  * Starting at offset uio->uio_loffset.
1129  */
1130 int
1131 dmu_read_uio(objset_t *os, uint64_t object, uio_t *uio, uint64_t size)
1132 {
1133         dnode_t *dn;
1134         int err;
```



```
1135
1136         if (size == 0)
1137                 return (0);
1138
1139         err = dnode_hold(os, object, FTAG, &dn);
1140         if (err)
1141                 return (err);
1142
1143         err = dmu_read_uio_dnode(dn, uio, size);
1144
1145         dnode_rele(dn, FTAG);
1146
1147         return (err);
1148 }
1149
1150 static int
1151 dmu_write_uio_dnode(dnode_t *dn, uio_t *uio, uint64_t size, dmu_tx_t *tx)
1152 {
1153         dmu_buf_t **dbp;
1154         int numbufs;
1155         int err = 0;
1156         int i;
1157
1158         err = dmu_buf_hold_array_by_dnode(dn, uio->uio_loffset, size,
1159             FALSE, FTAG, &numbufs, &dbp, DMU_READ_PREFETCH);
1160         if (err)
1161                 return (err);
1162
1163         for (i = 0; i < numbufs; i++) {
1164                 int tocpy;
1165                 int bufoff;
1166                 dmu_buf_t *db = dbp[i];
1167
1168                 ASSERT(size > 0);
1169
1170                 bufoff = uio->uio_loffset - db->db_offset;
1171                 tocpy = (int)MIN(db->db_size - bufoff, size);
1172
1173                 ASSERT(i == 0 || i == numbufs-1 || tocpy == db->db_size);
1174
1175                 if (tocpy == db->db_size)
1176                         dmu_buf_will_fill(db, tx);
1177                 else
1178                         dmu_buf_will_dirty(db, tx);
1179
1180                 /*
1181                  * XXX uiomove could block forever (eg. nfs-backed
1182                  * pages).  There needs to be a uiolockdown() function
1183                  * to lock the pages in memory, so that uiomove won't
1184                  * block.
1185                  */
1186                 err = uiomove((char *)db->db_data + bufoff, tocpy,
1187                     UIO_WRITE, uio);
1188
1189                 if (tocpy == db->db_size)
1190                         dmu_buf_fill_done(db, tx);
1191
1192                 if (err)
1193                         break;
1194
1195                 size -= tocpy;
1196         }
1197
1198         dmu_buf_rele_array(dbp, numbufs, FTAG);
1199         return (err);
1200 }
```

```
1202 /*
1203  * Write ’size’ bytes from the uio buffer.
1204  * To object zdb->db_object.
1205  * Starting at offset uio->uio_loffset.
1206  *
1207  * If the caller already has a dbuf in the target object
1208  * (e.g. its bonus buffer), this routine is faster than dmu_write_uio(),
1209  * because we don’t have to find the dnode_t for the object.
1210  */
1211 int
1212 dmu_write_uio_dbuf(dmu_buf_t *zdb, uio_t *uio, uint64_t size,
1213     dmu_tx_t *tx)
1214 {
1215         dmu_buf_impl_t *db = (dmu_buf_impl_t *)zdb;
1216         dnode_t *dn;
1217         int err;
1218
1219         if (size == 0)
1220                 return (0);
1221
1222         DB_DNODE_ENTER(db);
1223         dn = DB_DNODE(db);
1224         err = dmu_write_uio_dnode(dn, uio, size, tx);
1225         DB_DNODE_EXIT(db);
1226
1227         return (err);
1228 }
1229
1230 /*
1231  * Write ’size’ bytes from the uio buffer.
1232  * To the specified object.
1233  * Starting at offset uio->uio_loffset.
1234  */
1235 int
1236 dmu_write_uio(objset_t *os, uint64_t object, uio_t *uio, uint64_t size,
1237     dmu_tx_t *tx)
1238 {
1239         dnode_t *dn;
1240         int err;
1241
1242         if (size == 0)
1243                 return (0);
1244
1245         err = dnode_hold(os, object, FTAG, &dn);
1246         if (err)
1247                 return (err);
1248
1249         err = dmu_write_uio_dnode(dn, uio, size, tx);
1250
1251         dnode_rele(dn, FTAG);
1252
1253         return (err);
1254 }
1255
1256 int
1257 dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
1258     page_t *pp, dmu_tx_t *tx)
1259 {
1260         dmu_buf_t **dbp;
1261         int numbufs, i;
1262         int err;
1263
1264         if (size == 0)
1265                 return (0);
```



```
1267         err = dmu_buf_hold_array(os, object, offset, size,
1268             FALSE, FTAG, &numbufs, &dbp);
1269         if (err)
1270                 return (err);
1271
1272         for (i = 0; i < numbufs; i++) {
1273                 int tocpy, copied, thiscpy;
1274                 int bufoff;
1275                 dmu_buf_t *db = dbp[i];
1276                 caddr_t va;
1277
1278                 ASSERT(size > 0);
1279                 ASSERT3U(db->db_size, >=, PAGESIZE);
1280
1281                 bufoff = offset - db->db_offset;
1282                 tocpy = (int)MIN(db->db_size - bufoff, size);
1283
1284                 ASSERT(i == 0 || i == numbufs-1 || tocpy == db->db_size);
1285
1286                 if (tocpy == db->db_size)
1287                         dmu_buf_will_fill(db, tx);
1288                 else
1289                         dmu_buf_will_dirty(db, tx);
1290
1291                 for (copied = 0; copied < tocpy; copied += PAGESIZE) {
1292                         ASSERT3U(pp->p_offset, ==, db->db_offset + bufoff);
1293                         thiscpy = MIN(PAGESIZE, tocpy - copied);
1294                         va = zfs_map_page(pp, S_READ);
1295                         bcopy(va, (char *)db->db_data + bufoff, thiscpy);
1296                         zfs_unmap_page(pp, va);
1297                         pp = pp->p_next;
1298                         bufoff += PAGESIZE;
1299                 }
1300
1301                 if (tocpy == db->db_size)
1302                         dmu_buf_fill_done(db, tx);
1303
1304                 offset += tocpy;
1305                 size -= tocpy;
1306         }
1307         dmu_buf_rele_array(dbp, numbufs, FTAG);
1308         return (err);
1309 }
1310 #endif
1311
1312 /*
1313  * Allocate a loaned anonymous arc buffer.
1314  */
1315 arc_buf_t *
1316 dmu_request_arcbuf(dmu_buf_t *handle, int size)
1317 {
1318         dmu_buf_impl_t *db = (dmu_buf_impl_t *)handle;
1319
1320         return (arc_loan_buf(db->db_objset->os_spa, size));
1321 }
1322
1323 /*
1324  * Free a loaned arc buffer.
1325  */
1326 void
1327 dmu_return_arcbuf(arc_buf_t *buf)
1328 {
1329         arc_return_buf(buf, FTAG);
1330         VERIFY(arc_buf_remove_ref(buf, FTAG));
1331 }
```

```
1333 /*
1334  * When possible directly assign passed loaned arc buffer to a dbuf.
1335  * If this is not possible copy the contents of passed arc buf via
1336  * dmu_write().
1337  */
1338 void
1339 dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, arc_buf_t *buf,
1340     dmu_tx_t *tx)
1341 {
1342         dmu_buf_impl_t *dbuf = (dmu_buf_impl_t *)handle;
1343         dnode_t *dn;
1344         dmu_buf_impl_t *db;
1345         uint32_t blksz = (uint32_t)arc_buf_size(buf);
1346         uint64_t blkid;

1348         DB_DNODE_ENTER(dbuf);
1349         dn = DB_DNODE(dbuf);
1350         rw_enter(&dn->dn_struct_rwlock, RW_READER);
1351         blkid = dbuf_whichblock(dn, offset);
1352         VERIFY((db = dbuf_hold(dn, blkid, FTAG)) != NULL);
1353         rw_exit(&dn->dn_struct_rwlock);
1354         DB_DNODE_EXIT(dbuf);

1356         /*
1357          * We can only assign if the offset is aligned, the arc buf is the
1358          * same size as the dbuf, and the dbuf is not metadata.  It
1359          * can't be metadata because the loaned arc buf comes from the
1360          * user-data kmem arena.
1361          */
1362         if (offset == db->db.db_offset && blksz == db->db.db_size &&
1363             DBUF_GET_BUFC_TYPE(db) == ARC_BUFC_DATA) {
1364                 dbuf_assign_arcbuf(db, buf, tx);
1365                 dbuf_rele(db, FTAG);
1366         } else {
1367                 objset_t *os;
1368                 uint64_t object;

1370                 DB_DNODE_ENTER(dbuf);
1371                 dn = DB_DNODE(dbuf);
1372                 os = dn->dn_objset;
1373                 object = dn->dn_object;
1374                 DB_DNODE_EXIT(dbuf);

1376                 dbuf_rele(db, FTAG);
1377                 dmu_write(os, object, offset, blksz, buf->b_data, tx);
1378                 dmu_return_arcbuf(buf);
1379                 XUIOSTAT_BUMP(xuiostat_wbuf_copied);
1380         }
1381 }

1383 typedef struct {
1384         dbuf_dirty_record_t     *dsa_dr;
1385         dmu_sync_cb_t           *dsa_done;
1386         zgd_t                   *dsa_zgd;
1387         dmu_tx_t                *dsa_tx;
1388 } dmu_sync_arg_t;

1390 /* ARGSUSED */
1391 static void
1392 dmu_sync_ready(zio_t *zio, arc_buf_t *buf, void *varg)
1393 {
1394         dmu_sync_arg_t *dsa = varg;
1395         dmu_buf_t *db = dsa->dsa_zgd->zgd_db;
1396         blkptr_t *bp = zio->io_bp;

1398         if (zio->io_error == 0) {
```



```
1399                 if (BP_IS_HOLE(bp)) {
1400                         /*
1401                          * A block of zeros may compress to a hole, but the
1402                          * block size still needs to be known for replay.
1403                          */
1404                         BP_SET_LSIZE(bp, db->db_size);
1405                 } else if (!BP_IS_EMBEDDED(bp)) {
1406                         ASSERT(BP_GET_LEVEL(bp) == 0);
1407                         bp->blk_fill = 1;
1408                 }
1409         }
1410 }

1412 static void
1413 dmu_sync_late_arrival_ready(zio_t *zio)
1414 {
1415         dmu_sync_ready(zio, NULL, zio->io_private);
1416 }

1418 /* ARGSUSED */
1419 static void
1420 dmu_sync_done(zio_t *zio, arc_buf_t *buf, void *varg)
1421 {
1422         dmu_sync_arg_t *dsa = varg;
1423         dbuf_dirty_record_t *dr = dsa->dsa_dr;
1424         dmu_buf_impl_t *db = dr->dr_dbuf;

1426         mutex_enter(&db->db_mtx);
1427         ASSERT(dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC);
1428         if (zio->io_error == 0) {
1429                 dr->dt.dl.dr_nopwrite = !!(zio->io_flags & ZIO_FLAG_NOPWRITE);
1430                 if (dr->dt.dl.dr_nopwrite) {
1431                         blkptr_t *bp = zio->io_bp;
1432                         blkptr_t *bp_orig = &zio->io_bp_orig;
1433                         uint8_t chksum = BP_GET_CHECKSUM(bp_orig);

1435                         ASSERT(BP_EQUAL(bp, bp_orig));
1436                         ASSERT(zio->io_prop.zp_compress != ZIO_COMPRESS_OFF);
1437                         ASSERT(zio_checksum_table[chksum].ci_dedup);
1438                 }
1439                 dr->dt.dl.dr_overridden_by = *zio->io_bp;
1440                 dr->dt.dl.dr_override_state = DR_OVERRIDDEN;
1441                 dr->dt.dl.dr_copies = zio->io_prop.zp_copies;
1442                 if (BP_IS_HOLE(&dr->dt.dl.dr_overridden_by))
1443                         BP_ZERO(&dr->dt.dl.dr_overridden_by);
1444         } else {
1445                 dr->dt.dl.dr_override_state = DR_NOT_OVERRIDDEN;
1446         }
1447         cv_broadcast(&db->db_changed);
1448         mutex_exit(&db->db_mtx);

1450         dsa->dsa_done(dsa->dsa_zgd, zio->io_error);

1452         kmem_free(dsa, sizeof (*dsa));
1453 }

1455 static void
1456 dmu_sync_late_arrival_done(zio_t *zio)
1457 {
1458         blkptr_t *bp = zio->io_bp;
1459         dmu_sync_arg_t *dsa = zio->io_private;
1460         blkptr_t *bp_orig = &zio->io_bp_orig;

1462         if (zio->io_error == 0 && !BP_IS_HOLE(bp)) {
1463                 /*
1464                  * If we didn't allocate a new block (i.e. ZIO_FLAG_NOPWRITE)
```

```
1465                  * then there is nothing to do here. Otherwise, free the
1466                  * newly allocated block in this txg.
1467                  */
1468                 if (zio->io_flags & ZIO_FLAG_NOPWRITE) {
1469                         ASSERT(BP_EQUAL(bp, bp_orig));
1470                 } else {
1471                         ASSERT(BP_IS_HOLE(bp_orig) || !BP_EQUAL(bp, bp_orig));
1472                         ASSERT(zio->io_bp->blk_birth == zio->io_txg);
1473                         ASSERT(zio->io_txg > spa_syncing_txg(zio->io_spa));
1474                         zio_free(zio->io_spa, zio->io_txg, zio->io_bp);
1475                 }
1476         }

1478         dmu_tx_commit(dsa->dsa_tx);

1480         dsa->dsa_done(dsa->dsa_zgd, zio->io_error);

1482         kmem_free(dsa, sizeof (*dsa));
1483 }

1485 static int
1486 dmu_sync_late_arrival(zio_t *pio, objset_t *os, dmu_sync_cb_t *done, zgd_t *zgd,
1487     zio_prop_t *zp, zbookmark_phys_t *zb)
1488 {
1489         dmu_sync_arg_t *dsa;
1490         dmu_tx_t *tx;

1492         tx = dmu_tx_create(os);
1493         dmu_tx_hold_space(tx, zgd->zgd_db->db_size);
1494         if (dmu_tx_assign(tx, TXG_WAIT) != 0) {
1495                 dmu_tx_abort(tx);
1496                 /* Make zl_get_data do txg_waited_synced() */
1497                 return (SET_ERROR(EIO));
1498         }

1500         dsa = kmem_alloc(sizeof (dmu_sync_arg_t), KM_SLEEP);
1501         dsa->dsa_dr = NULL;
1502         dsa->dsa_done = done;
1503         dsa->dsa_zgd = zgd;
1504         dsa->dsa_tx = tx;

1506         zio_nowait(zio_write(pio, os->os_spa, dmu_tx_get_txg(tx), zgd->zgd_bp,
1507             zgd->zgd_db->db_data, zgd->zgd_db->db_size, zp,
1508             dmu_sync_late_arrival_ready, NULL, dmu_sync_late_arrival_done, dsa,
1509             ZIO_PRIORITY_SYNC_WRITE, ZIO_FLAG_CANFAIL, zb));

1511         return (0);
1512 }

1514 /*
1515  * Intent log support: sync the block associated with db to disk.
1516  * N.B. and XXX: the caller is responsible for making sure that the
1517  * data isn't changing while dmu_sync() is writing it.
1518  *
1519  * Return values:
1520  *
1521  *      EEXIST: this txg has already been synced, so there's nothing to do.
1522  *              The caller should not log the write.
1523  *
1524  *      ENOENT: the block was dbuf_free_range()'d, so there's nothing to do.
1525  *              The caller should not log the write.
1526  *
1527  *      EALREADY: this block is already in the process of being synced.
1528  *              The caller should track its progress (somehow).
1529  *
1530  *      EIO: could not do the I/O.
```



```
1531  *              The caller should do a txg_wait_synced().
1532  *
1533  *      0: the I/O has been initiated.
1534  *              The caller should log this blkptr in the done callback.
1535  *              It is possible that the I/O will fail, in which case
1536  *              the error will be reported to the done callback and
1537  *              propagated to pio from zio_done().
1538  */
1539 int
1540 dmu_sync(zio_t *pio, uint64_t txg, dmu_sync_cb_t *done, zgd_t *zgd)
1541 {
1542         blkptr_t *bp = zgd->zgd_bp;
1543         dmu_buf_impl_t *db = (dmu_buf_impl_t *)zgd->zgd_db;
1544         objset_t *os = db->db_objset;
1545         dsl_dataset_t *ds = os->os_dsl_dataset;
1546         dbuf_dirty_record_t *dr;
1547         dmu_sync_arg_t *dsa;
1548         zbookmark_phys_t zb;
1549         zio_prop_t zp;
1550         dnode_t *dn;

1552         ASSERT(pio != NULL);
1553         ASSERT(txg != 0);

1555         SET_BOOKMARK(&zb, ds->ds_object,
1556             db->db.db_object, db->db_level, db->db_blkid);

1558         DB_DNODE_ENTER(db);
1559         dn = DB_DNODE(db);
1560         dmu_write_policy(os, dn, db->db_level, WP_DMU_SYNC, &zp);
1561         DB_DNODE_EXIT(db);

1563         /*
1564          * If we're frozen (running ziltest), we always need to generate a bp.
1565          */
1566         if (txg > spa_freeze_txg(os->os_spa))
1567                 return (dmu_sync_late_arrival(pio, os, done, zgd, &zp, &zb));

1569         /*
1570          * Grabbing db_mtx now provides a barrier between dbuf_sync_leaf()
1571          * and us.  If we determine that this txg is not yet syncing,
1572          * but it begins to sync a moment later, that's OK because the
1573          * sync thread will block in dbuf_sync_leaf() until we drop db_mtx.
1574          */
1575         mutex_enter(&db->db_mtx);

1577         if (txg <= spa_last_synced_txg(os->os_spa)) {
1578                 /*
1579                  * This txg has already synced.  There's nothing to do.
1580                  */
1581                 mutex_exit(&db->db_mtx);
1582                 return (SET_ERROR(EEXIST));
1583         }

1585         if (txg <= spa_syncing_txg(os->os_spa)) {
1586                 /*
1587                  * This txg is currently syncing, so we can't mess with
1588                  * the dirty record anymore; just write a new log block.
1589                  */
1590                 mutex_exit(&db->db_mtx);
1591                 return (dmu_sync_late_arrival(pio, os, done, zgd, &zp, &zb));
1592         }

1594         dr = db->db_last_dirty;
1595         while (dr && dr->dr_txg != txg)
1596                 dr = dr->dr_next;
```

```
1598         if (dr == NULL) {
1599                 /*
1600                  * There's no dr for this dbuf, so it must have been freed.
1601                  * There's no need to log writes to freed blocks, so we're done.
1602                  */
1603                 mutex_exit(&db->db_mtx);
1604                 return (SET_ERROR(ENOENT));
1605         }

1607         ASSERT(dr->dr_next == NULL || dr->dr_next->dr_txg < txg);

1609         /*
1610          * Assume the on-disk data is X, the current syncing data is Y,
1611          * and the current in-memory data is Z (currently in dmu_sync).
1612          * X and Z are identical but Y is has been modified. Normally,
1613          * when X and Z are the same we will perform a nopwrite but if Y
1614          * is different we must disable nopwrite since the resulting write
1615          * of Y to disk can free the block containing X. If we allowed a
1616          * nopwrite to occur the block pointing to Z would reference a freed
1617          * block. Since this is a rare case we simplify this by disabling
1618          * nopwrite if the current dmu_sync-ing dbuf has been modified in
1619          * a previous transaction.
1620          */
1621         if (dr->dr_next)
1622                 zp.zp_nopwrite = B_FALSE;

1624         ASSERT(dr->dr_txg == txg);
1625         if (dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC ||
1626             dr->dt.dl.dr_override_state == DR_OVERRIDDEN) {
1627                 /*
1628                  * We have already issued a sync write for this buffer,
1629                  * or this buffer has already been synced.  It could not
1630                  * have been dirtied since, or we would have cleared the state.
1631                  */
1632                 mutex_exit(&db->db_mtx);
1633                 return (SET_ERROR(EALREADY));
1634         }

1636         ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
1637         dr->dt.dl.dr_override_state = DR_IN_DMU_SYNC;
1638         mutex_exit(&db->db_mtx);

1640         dsa = kmem_alloc(sizeof (dmu_sync_arg_t), KM_SLEEP);
1641         dsa->dsa_dr = dr;
1642         dsa->dsa_done = done;
1643         dsa->dsa_zgd = zgd;
1644         dsa->dsa_tx = NULL;

1646         zio_nowait(arc_write(pio, os->os_spa, txg,
1647             bp, dr->dt.dl.dr_data, DBUF_IS_L2CACHEABLE(db),
1648             DBUF_IS_L2COMPRESSIBLE(db), &zp, dmu_sync_ready,
1649             NULL, dmu_sync_done, dsa, ZIO_PRIORITY_SYNC_WRITE,
1650             ZIO_FLAG_CANFAIL, &zb));

1652         return (0);
1653 }

1655 int
1656 dmu_object_set_blocksize(objset_t *os, uint64_t object, uint64_t size, int ibs,
1657         dmu_tx_t *tx)
1658 {
1659         dnode_t *dn;
1660         int err;

1662         err = dnode_hold(os, object, FTAG, &dn);
```



```
1663         if (err)
1664                 return (err);
1665         err = dnode_set_blksz(dn, size, ibs, tx);
1666         dnode_rele(dn, FTAG);
1667         return (err);
1668 }

1670 void
1671 dmu_object_set_checksum(objset_t *os, uint64_t object, uint8_t checksum,
1672         dmu_tx_t *tx)
1673 {
1674         dnode_t *dn;

1676         /*
1677          * Send streams include each object's checksum function.  This
1678          * check ensures that the receiving system can understand the
1679          * checksum function transmitted.
1680          */
1681         ASSERT3U(checksum, <, ZIO_CHECKSUM_LEGACY_FUNCTIONS);

1683         VERIFY0(dnode_hold(os, object, FTAG, &dn));
1684         ASSERT3U(checksum, <, ZIO_CHECKSUM_FUNCTIONS);
1685         dn->dn_checksum = checksum;
1686         dnode_setdirty(dn, tx);
1687         dnode_rele(dn, FTAG);
1688 }

1690 void
1691 dmu_object_set_compress(objset_t *os, uint64_t object, uint8_t compress,
1692         dmu_tx_t *tx)
1693 {
1694         dnode_t *dn;

1696         /*
1697          * Send streams include each object's compression function.  This
1698          * check ensures that the receiving system can understand the
1699          * compression function transmitted.
1700          */
1701         ASSERT3U(compress, <, ZIO_COMPRESS_LEGACY_FUNCTIONS);

1703         VERIFY0(dnode_hold(os, object, FTAG, &dn));
1704         dn->dn_compress = compress;
1705         dnode_setdirty(dn, tx);
1706         dnode_rele(dn, FTAG);
1707 }

1709 int zfs_mdcomp_disable = 0;

1711 /*
1712  * When the "redundant_metadata" property is set to "most", only indirect
1713  * blocks of this level and higher will have an additional ditto block.
1714  */
1715 int zfs_redundant_metadata_most_ditto_level = 2;

1717 void
1718 dmu_write_policy(objset_t *os, dnode_t *dn, int level, int wp, zio_prop_t *zp)
1719 {
1720         dmu_object_type_t type = dn ? dn->dn_type : DMU_OT_OBJSET;
1721         boolean_t ismd = (level > 0 || DMU_OT_IS_METADATA(type) ||
1722             (wp & WP_SPILL));
1723         enum zio_checksum checksum = os->os_checksum;
1724         enum zio_compress compress = os->os_compress;
1725         enum zio_checksum dedup_checksum = os->os_dedup_checksum;
1726         boolean_t dedup = B_FALSE;
1727         boolean_t nopwrite = B_FALSE;
1728         boolean_t dedup_verify = os->os_dedup_verify;
```

```
1729         int copies = os->os_copies;

1731         /*
1732          * We maintain different write policies for each of the following
1733          * types of data:
1734          *       1. metadata
1735          *       2. preallocated blocks (i.e. level-0 blocks of a dump device)
1736          *       3. all other level 0 blocks
1737          */
1738         if (ismd) {
1739                 /*
1740                  * XXX -- we should design a compression algorithm
1741                  * that specializes in arrays of bps.
1742                  */
1743                 boolean_t lz4_ac = spa_feature_is_active(os->os_spa,
1744                     SPA_FEATURE_LZ4_COMPRESS);

1746                 if (zfs_mdcomp_disable) {
1747                         compress = ZIO_COMPRESS_EMPTY;
1748                 } else if (lz4_ac) {
1749                         compress = ZIO_COMPRESS_LZ4;
1750                 } else {
1751                         compress = ZIO_COMPRESS_LZJB;
1752                 }

1754                 /*
1755                  * Metadata always gets checksummed.  If the data
1756                  * checksum is multi-bit correctable, and it's not a
1757                  * ZBT-style checksum, then it's suitable for metadata
1758                  * as well.  Otherwise, the metadata checksum defaults
1759                  * to fletcher4.
1760                  */
1761                 if (zio_checksum_table[checksum].ci_correctable < 1 ||
1762                     zio_checksum_table[checksum].ci_eck)
1763                         checksum = ZIO_CHECKSUM_FLETCHER_4;

1765                 if (os->os_redundant_metadata == ZFS_REDUNDANT_METADATA_ALL ||
1766                     (os->os_redundant_metadata ==
1767                     ZFS_REDUNDANT_METADATA_MOST &&
1768                     (level >= zfs_redundant_metadata_most_ditto_level ||
1769                     DMU_OT_IS_METADATA(type) || (wp & WP_SPILL))))
1770                         copies++;
1771         } else if (wp & WP_NOFILL) {
1772                 ASSERT(level == 0);

1774                 /*
1775                  * If we're writing preallocated blocks, we aren't actually
1776                  * writing them so don't set any policy properties.  These
1777                  * blocks are currently only used by an external subsystem
1778                  * outside of zfs (i.e. dump) and not written by the zio
1779                  * pipeline.
1780                  */
1781                 compress = ZIO_COMPRESS_OFF;
1782                 checksum = ZIO_CHECKSUM_NOPARITY;
1783         } else {
1784                 compress = zio_compress_select(dn->dn_compress, compress);

1786                 checksum = (dedup_checksum == ZIO_CHECKSUM_OFF) ?
1787                     zio_checksum_select(dn->dn_checksum, checksum) :
1788                     dedup_checksum;

1790                 /*
1791                  * Determine dedup setting.  If we are in dmu_sync(),
1792                  * we won't actually dedup now because that's all
1793                  * done in syncing context; but we do want to use the
1794                  * dedup checkum.  If the checksum is not strong
```



```
1795                  * enough to ensure unique signatures, force
1796                  * dedup_verify.
1797                  */
1798                 if (dedup_checksum != ZIO_CHECKSUM_OFF) {
1799                         dedup = (wp & WP_DMU_SYNC) ? B_FALSE : B_TRUE;
1800                         if (!zio_checksum_table[checksum].ci_dedup)
1801                                 dedup_verify = B_TRUE;
1802                 }

1804                 /*
1805                  * Enable nopwrite if we have a cryptographically secure
1806                  * checksum that has no known collisions (i.e. SHA-256)
1807                  * and compression is enabled.  We don't enable nopwrite if
1808                  * dedup is enabled as the two features are mutually exclusive.
1809                  */
1810                 nopwrite = (!dedup && zio_checksum_table[checksum].ci_dedup &&
1811                     compress != ZIO_COMPRESS_OFF && zfs_nopwrite_enabled);
1812         }

1814         zp->zp_checksum = checksum;
1815         zp->zp_compress = compress;
1816         zp->zp_type = (wp & WP_SPILL) ? dn->dn_bonustype : type;
1817         zp->zp_level = level;
1818         zp->zp_copies = MIN(copies, spa_max_replication(os->os_spa));
1819         zp->zp_dedup = dedup;
1820         zp->zp_dedup_verify = dedup && dedup_verify;
1821         zp->zp_nopwrite = nopwrite;
1822         zp->zp_zero_write = B_FALSE;
1823 #endif /* ! codereview */
1824 }

1826 int
1827 dmu_offset_next(objset_t *os, uint64_t object, boolean_t hole, uint64_t *off)
1828 {
1829         dnode_t *dn;
1830         int i, err;

1832         err = dnode_hold(os, object, FTAG, &dn);
1833         if (err)
1834                 return (err);
1835         /*
1836          * Sync any current changes before
1837          * we go trundling through the block pointers.
1838          */
1839         for (i = 0; i < TXG_SIZE; i++) {
1840                 if (list_link_active(&dn->dn_dirty_link[i]))
1841                         break;
1842         }
1843         if (i != TXG_SIZE) {
1844                 dnode_rele(dn, FTAG);
1845                 txg_wait_synced(dmu_objset_pool(os), 0);
1846                 err = dnode_hold(os, object, FTAG, &dn);
1847                 if (err)
1848                         return (err);
1849         }

1851         err = dnode_next_offset(dn, (hole ? DNODE_FIND_HOLE : 0), off, 1, 1, 0);
1852         dnode_rele(dn, FTAG);

1854         return (err);
1855 }

1857 void
1858 dmu_object_info_from_dnode(dnode_t *dn, dmu_object_info_t *doi)
1859 {
1860         dnode_phys_t *dnp;
```

```
1862         rw_enter(&dn->dn_struct_rwlock, RW_READER);
1863         mutex_enter(&dn->dn_mtx);

1865         dnp = dn->dn_phys;

1867         doi->doi_data_block_size = dn->dn_datablksz;
1868         doi->doi_metadata_block_size = dn->dn_indblkshift ?
1869             1ULL << dn->dn_indblkshift : 0;
1870         doi->doi_type = dn->dn_type;
1871         doi->doi_bonus_type = dn->dn_bonustype;
1872         doi->doi_bonus_size = dn->dn_bonuslen;
1873         doi->doi_indirection = dn->dn_nlevels;
1874         doi->doi_checksum = dn->dn_checksum;
1875         doi->doi_compress = dn->dn_compress;
1876         doi->doi_nblkptr = dn->dn_nblkptr;
1877         doi->doi_physical_blocks_512 = (DN_USED_BYTES(dnp) + 256) >> 9;
1878         doi->doi_max_offset = (dn->dn_maxblkid + 1) * dn->dn_datablksz;
1879         doi->doi_fill_count = 0;
1880         for (int i = 0; i < dnp->dn_nblkptr; i++)
1881                 doi->doi_fill_count += BP_GET_FILL(&dnp->dn_blkptr[i]);

1883         mutex_exit(&dn->dn_mtx);
1884         rw_exit(&dn->dn_struct_rwlock);
1885 }

1887 /*
1888  * Get information on a DMU object.
1889  * If doi is NULL, just indicates whether the object exists.
1890  */
1891 int
1892 dmu_object_info(objset_t *os, uint64_t object, dmu_object_info_t *doi)
1893 {
1894         dnode_t *dn;
1895         int err = dnode_hold(os, object, FTAG, &dn);

1897         if (err)
1898                 return (err);

1900         if (doi != NULL)
1901                 dmu_object_info_from_dnode(dn, doi);

1903         dnode_rele(dn, FTAG);
1904         return (0);
1905 }

1907 /*
1908  * As above, but faster; can be used when you have a held dbuf in hand.
1909  */
1910 void
1911 dmu_object_info_from_db(dmu_buf_t *db_fake, dmu_object_info_t *doi)
1912 {
1913         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;

1915         DB_DNODE_ENTER(db);
1916         dmu_object_info_from_dnode(DB_DNODE(db), doi);
1917         DB_DNODE_EXIT(db);
1918 }

1920 /*
1921  * Faster still when you only care about the size.
1922  * This is specifically optimized for zfs_getattr().
1923  */
1924 void
1925 dmu_object_size_from_db(dmu_buf_t *db_fake, uint32_t *blksize,
1926     u_longlong_t *nblk512)
```



```
1927 {
1928         dmu_buf_impl_t *db = (dmu_buf_impl_t *)db_fake;
1929         dnode_t *dn;

1931         DB_DNODE_ENTER(db);
1932         dn = DB_DNODE(db);

1934         *blksize = dn->dn_datablksz;
1935         /* add 1 for dnode space */
1936         *nblk512 = ((DN_USED_BYTES(dn->dn_phys) + SPA_MINBLOCKSIZE/2) >>
1937             SPA_MINBLOCKSHIFT) + 1;
1938         DB_DNODE_EXIT(db);
1939 }

1941 void
1942 byteswap_uint64_array(void *vbuf, size_t size)
1943 {
1944         uint64_t *buf = vbuf;
1945         size_t count = size >> 3;
1946         int i;

1948         ASSERT((size & 7) == 0);

1950         for (i = 0; i < count; i++)
1951                 buf[i] = BSWAP_64(buf[i]);
1952 }

1954 void
1955 byteswap_uint32_array(void *vbuf, size_t size)
1956 {
1957         uint32_t *buf = vbuf;
1958         size_t count = size >> 2;
1959         int i;

1961         ASSERT((size & 3) == 0);

1963         for (i = 0; i < count; i++)
1964                 buf[i] = BSWAP_32(buf[i]);
1965 }

1967 void
1968 byteswap_uint16_array(void *vbuf, size_t size)
1969 {
1970         uint16_t *buf = vbuf;
1971         size_t count = size >> 1;
1972         int i;

1974         ASSERT((size & 1) == 0);

1976         for (i = 0; i < count; i++)
1977                 buf[i] = BSWAP_16(buf[i]);
1978 }

1980 /* ARGSUSED */
1981 void
1982 byteswap_uint8_array(void *vbuf, size_t size)
1983 {
1984 }

1986 void
1987 dmu_init(void)
1988 {
1989         zfs_dbgmsg_init();
1990         sa_cache_init();
1991         xuio_stat_init();
1992         dmu_objset_init();
```

```
1993         dnode_init();
1994         dbuf_init();
1995         zfetch_init();
1996         l2arc_init();
1997         arc_init();
1998 }

2000 void
2001 dmu_fini(void)
2002 {
2003         arc_fini(); /* arc depends on l2arc, so arc must go first */
2004         l2arc_fini();
2005         zfetch_fini();
2006         dbuf_fini();
2007         dnode_fini();
2008         dmu_objset_fini();
2009         xuio_stat_fini();
2010         sa_cache_fini();
2011         zfs_dbgmsg_fini();
2012 }
```

```
**********************************************************
   55122 Tue Oct 28 11:57:19 2014
new/usr/src/uts/common/fs/zfs/dnode.c
Possibility to physically reserve space without writing leaf blocks
**********************************************************
______unchanged_portion_omitted_

1238 void
1239 dnode_setdirty(dnode_t *dn, dmu_tx_t *tx)
1240 {
1241         objset_t *os = dn->dn_objset;
1242         uint64_t txg = tx->tx_txg;

1244         if (DMU_OBJECT_IS_SPECIAL(dn->dn_object)) {
1245                 dsl_dataset_dirty(os->os_dsl_dataset, tx);
1246                 return;
1247         }

1249         DNODE_VERIFY(dn);

1251 #ifdef ZFS_DEBUG
1252         mutex_enter(&dn->dn_mtx);
1253         ASSERT(dn->dn_phys->dn_type || dn->dn_allocated_txg);
1254         ASSERT(dn->dn_free_txg == 0 || dn->dn_free_txg >= txg);
1255         mutex_exit(&dn->dn_mtx);
1256 #endif

1258         /*
1259          * Determine old uid/gid when necessary
1260          */
1261         dmu_objset_userquota_get_ids(dn, B_TRUE, tx);

1263         mutex_enter(&os->os_lock);

1265         /*
1266          * If we are already marked dirty, we're done.
1267          */
1268         if (list_link_active(&dn->dn_dirty_link[txg & TXG_MASK])) {
1269                 mutex_exit(&os->os_lock);
1270                 return;
1271         }

1273         ASSERT(!refcount_is_zero(&dn->dn_holds) ||
1274             !avl_is_empty(&dn->dn_dbufs));
1275         ASSERT(dn->dn_datablksz != 0);
1276         ASSERT0(dn->dn_next_bonuslen[txg&TXG_MASK]);
1277         ASSERT0(dn->dn_next_blksz[txg&TXG_MASK]);
1278         ASSERT0(dn->dn_next_bonustype[txg&TXG_MASK]);

1280         dprintf_ds(os->os_dsl_dataset, "obj=%llu txg=%llu\n",
1281             dn->dn_object, txg);

1283         if (dn->dn_free_txg > 0 && dn->dn_free_txg <= txg) {
1284                 list_insert_tail(&os->os_free_dnodes[txg&TXG_MASK], dn);
1285         } else {
1286                 list_insert_tail(&os->os_dirty_dnodes[txg&TXG_MASK], dn);
1287         }

1289         mutex_exit(&os->os_lock);

1291         /*
1292          * The dnode maintains a hold on its containing dbuf as
1293          * long as there are holds on it.  Each instantiated child
1294          * dbuf maintains a hold on the dnode.  When the last child
1295          * drops its hold, the dnode will drop its hold on the
1296          * containing dbuf. We add a "dirty hold" here so that the
```



```
1297          * dnode will hang around after we finish processing its
1298          * children.
1299          */
1300         VERIFY(dnode_add_ref(dn, (void *)(uintptr_t)tx->tx_txg));

1302         (void) dbuf_dirty(dn->dn_dbuf, tx, B_FALSE);
1302         (void) dbuf_dirty(dn->dn_dbuf, tx);

1304         dsl_dataset_dirty(os->os_dsl_dataset, tx);
1305 }
______unchanged_portion_omitted_

1409 /* read-holding callers must not rely on the lock being continuously held */
1410 void
1411 dnode_new_blkid(dnode_t *dn, uint64_t blkid, dmu_tx_t *tx, boolean_t have_read)
1412 {
1413         uint64_t txgoff = tx->tx_txg & TXG_MASK;
1414         int epbs, new_nlevels;
1415         uint64_t sz;

1417         ASSERT(blkid != DMU_BONUS_BLKID);

1419         ASSERT(have_read ?
1420             RW_READ_HELD(&dn->dn_struct_rwlock) :
1421             RW_WRITE_HELD(&dn->dn_struct_rwlock));

1423         /*
1424          * if we have a read-lock, check to see if we need to do any work
1425          * before upgrading to a write-lock.
1426          */
1427         if (have_read) {
1428                 if (blkid <= dn->dn_maxblkid)
1429                         return;

1431                 if (!rw_tryupgrade(&dn->dn_struct_rwlock)) {
1432                         rw_exit(&dn->dn_struct_rwlock);
1433                         rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1434                 }
1435         }

1437         if (blkid <= dn->dn_maxblkid)
1438                 goto out;

1440         dn->dn_maxblkid = blkid;

1442         /*
1443          * Compute the number of levels necessary to support the new maxblkid.
1444          */
1445         new_nlevels = 1;
1446         epbs = dn->dn_indblkshift - SPA_BLKPTRSHIFT;
1447         for (sz = dn->dn_nblkptr;
1448             sz <= blkid && sz >= dn->dn_nblkptr; sz <<= epbs)
1449                 new_nlevels++;

1451         if (new_nlevels > dn->dn_nlevels) {
1452                 int old_nlevels = dn->dn_nlevels;
1453                 dmu_buf_impl_t *db;
1454                 list_t *list;
1455                 dbuf_dirty_record_t *new, *dr, *dr_next;

1457                 dn->dn_nlevels = new_nlevels;

1459                 ASSERT3U(new_nlevels, >, dn->dn_next_nlevels[txgoff]);
1460                 dn->dn_next_nlevels[txgoff] = new_nlevels;

1462                 /* dirty the left indirects */
```

```
1463                 db = dbuf_hold_level(dn, old_nlevels, 0, FTAG);
1464                 ASSERT(db != NULL);
1465                 new = dbuf_dirty(db, tx, B_FALSE);
1465                 new = dbuf_dirty(db, tx);
1466                 dbuf_rele(db, FTAG);

1468                 /* transfer the dirty records to the new indirect */
1469                 mutex_enter(&dn->dn_mtx);
1470                 mutex_enter(&new->dt.di.dr_mtx);
1471                 list = &dn->dn_dirty_records[txgoff];
1472                 for (dr = list_head(list); dr; dr = dr_next) {
1473                         dr_next = list_next(&dn->dn_dirty_records[txgoff], dr);
1474                         if (dr->dr_dbuf->db_level != new_nlevels-1 &&
1475                             dr->dr_dbuf->db_blkid != DMU_BONUS_BLKID &&
1476                             dr->dr_dbuf->db_blkid != DMU_SPILL_BLKID) {
1477                                 ASSERT(dr->dr_dbuf->db_level == old_nlevels-1);
1478                                 list_remove(&dn->dn_dirty_records[txgoff], dr);
1479                                 list_insert_tail(&new->dt.di.dr_children, dr);
1480                                 dr->dr_parent = new;
1481                         }
1482                 }
1483                 mutex_exit(&new->dt.di.dr_mtx);
1484                 mutex_exit(&dn->dn_mtx);
1485         }

1487 out:
1488         if (have_read)
1489                 rw_downgrade(&dn->dn_struct_rwlock);
1490 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   10506 Tue Oct 28 11:57:19 2014
new/usr/src/uts/common/fs/zfs/sys/dbuf.h
Possibility to physically reserve space without writing leaf blocks
**********************************************************
______unchanged_portion_omitted_

typedef struct dbuf_dirty_record {
        /* link on our parents dirty list */
        list_node_t dr_dirty_node;

        /* transaction group this data will sync in */
        uint64_t dr_txg;

        /* zio of outstanding write IO */
        zio_t *dr_zio;

        /* pointer back to our dbuf */
        struct dmu_buf_impl *dr_dbuf;

        /* pointer to next dirty record */
        struct dbuf_dirty_record *dr_next;

        /* pointer to parent dirty record */
        struct dbuf_dirty_record *dr_parent;

        /* How much space was changed to dsl_pool_dirty_space() for this? */
        unsigned int dr_accounted;

        union dirty_types {
                struct dirty_indirect {

                        /* protect access to list */
                        kmutex_t dr_mtx;

                        /* Our list of dirty children */
                        list_t dr_children;
                } di;
                struct dirty_leaf {

                        /*
                         * dr_data is set when we dirty the buffer
                         * so that we can retain the pointer even if it
                         * gets COW'd in a subsequent transaction group.
                         */
                        arc_buf_t *dr_data;
                        blkptr_t dr_overridden_by;
                        override_states_t dr_override_state;
                        uint8_t dr_copies;
                        boolean_t dr_nopwrite;
                } dl;
        } dt;

        boolean_t dr_zero_write;
#endif /* ! codereview */
} dbuf_dirty_record_t;

typedef struct dmu_buf_impl {
        /*
         * The following members are immutable, with the exception of
         * db.db_data, which is protected by db_mtx.
         */

        /* the publicly visible structure */
        dmu_buf_t db;
```



```
        /* the objset we belong to */
        struct objset *db_objset;

        /*
         * handle to safely access the dnode we belong to (NULL when evicted)
         */
        struct dnode_handle *db_dnode_handle;

        /*
         * our parent buffer; if the dnode points to us directly,
         * db_parent == db_dnode_handle->dnh_dnode->dn_dbuf
         * only accessed by sync thread ???
         * (NULL when evicted)
         * May change from NULL to non-NULL under the protection of db_mtx
         * (see dbuf_check_blkptr())
         */
        struct dmu_buf_impl *db_parent;

        /*
         * link for hash table of all dmu_buf_impl_t's
         */
        struct dmu_buf_impl *db_hash_next;

        /* our block number */
        uint64_t db_blkid;

        /*
         * Pointer to the blkptr_t which points to us. May be NULL if we
         * don't have one yet. (NULL when evicted)
         */
        blkptr_t *db_blkptr;

        /*
         * Our indirection level.  Data buffers have db_level==0.
         * Indirect buffers which point to data buffers have
         * db_level==1. etc.  Buffers which contain dnodes have
         * db_level==0, since the dnodes are stored in a file.
         */
        uint8_t db_level;

        /* db_mtx protects the members below */
        kmutex_t db_mtx;

        /*
         * Current state of the buffer
         */
        dbuf_states_t db_state;

        /*
         * Refcount accessed by dmu_buf_{hold,rele}.
         * If nonzero, the buffer can't be destroyed.
         * Protected by db_mtx.
         */
        refcount_t db_holds;

        /* buffer holding our data */
        arc_buf_t *db_buf;

        kcondvar_t db_changed;
        dbuf_dirty_record_t *db_data_pending;

        /* pointer to most recent dirty record for this buffer */
        dbuf_dirty_record_t *db_last_dirty;

        /*
         * Our link on the owner dnodes's dn_dbufs list.
```

```
226          * Protected by its dn_dbufs_mtx.
227          */
228         avl_node_t db_link;

230         /* Data which is unique to data (leaf) blocks: */

232         /* stuff we store for the user (see dmu_buf_set_user) */
233         void *db_user_ptr;
234         void **db_user_data_ptr_ptr;
235         dmu_buf_evict_func_t *db_evict_func;

237         uint8_t db_immediate_evict;
238         uint8_t db_freed_in_flight;

240         uint8_t db_dirtycnt;
241 } dmu_buf_impl_t;

243 /* Note: the dbuf hash table is exposed only for the mdb module */
244 #define DBUF_MUTEXES 256
245 #define DBUF_HASH_MUTEX(h, idx) (&(h)->hash_mutexes[(idx) & (DBUF_MUTEXES-1)])
246 typedef struct dbuf_hash_table {
247         uint64_t hash_table_mask;
248         dmu_buf_impl_t **hash_table;
249         kmutex_t hash_mutexes[DBUF_MUTEXES];
250 } dbuf_hash_table_t;


253 uint64_t dbuf_whichblock(struct dnode *di, uint64_t offset);

255 dmu_buf_impl_t *dbuf_create_tlib(struct dnode *dn, char *data);
256 void dbuf_create_bonus(struct dnode *dn);
257 int dbuf_spill_set_blksz(dmu_buf_t *db, uint64_t blksz, dmu_tx_t *tx);
258 void dbuf_spill_hold(struct dnode *dn, dmu_buf_impl_t **dbp, void *tag);

260 void dbuf_rm_spill(struct dnode *dn, dmu_tx_t *tx);

262 dmu_buf_impl_t *dbuf_hold(struct dnode *dn, uint64_t blkid, void *tag);
263 dmu_buf_impl_t *dbuf_hold_level(struct dnode *dn, int level, uint64_t blkid,
264     void *tag);
265 int dbuf_hold_impl(struct dnode *dn, uint8_t level, uint64_t blkid, int create,
266     void *tag, dmu_buf_impl_t **dbp);

268 void dbuf_prefetch(struct dnode *dn, uint64_t blkid, zio_priority_t prio);

270 void dbuf_add_ref(dmu_buf_impl_t *db, void *tag);
271 uint64_t dbuf_refcount(dmu_buf_impl_t *db);

273 void dbuf_rele(dmu_buf_impl_t *db, void *tag);
274 void dbuf_rele_and_unlock(dmu_buf_impl_t *db, void *tag);

276 dmu_buf_impl_t *dbuf_find(struct dnode *dn, uint8_t level, uint64_t blkid);

278 int dbuf_read(dmu_buf_impl_t *db, zio_t *zio, uint32_t flags);
279 void dmu_buf_will_not_fill(dmu_buf_t *db, dmu_tx_t *tx);
280 void dmu_buf_will_fill(dmu_buf_t *db, dmu_tx_t *tx);
281 void dmu_buf_will_zero_fill(dmu_buf_t *db, dmu_tx_t *tx);
282 #endif /* ! codereview */
283 void dmu_buf_fill_done(dmu_buf_t *db, dmu_tx_t *tx);
284 void dbuf_assign_arcbuf(dmu_buf_impl_t *db, arc_buf_t *buf, dmu_tx_t *tx);
285 dbuf_dirty_record_t *dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx, boolean_t zero
286 dbuf_dirty_record_t *dbuf_zero_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
146 dbuf_dirty_record_t *dbuf_dirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
287 arc_buf_t *dbuf_loan_arcbuf(dmu_buf_impl_t *db);
288 void dmu_buf_write_embedded(dmu_buf_t *dbuf, void *data,
289     bp_embedded_type_t etype, enum zio_compress comp,
290     int uncompressed_size, int compressed_size, int byteorder, dmu_tx_t *tx);
```



```
292 void dbuf_clear(dmu_buf_impl_t *db);
293 void dbuf_evict(dmu_buf_impl_t *db);

295 void dbuf_setdirty(dmu_buf_impl_t *db, dmu_tx_t *tx);
296 void dbuf_unoverride(dbuf_dirty_record_t *dr);
297 void dbuf_sync_list(list_t *list, dmu_tx_t *tx);
298 void dbuf_release_bp(dmu_buf_impl_t *db);

300 void dbuf_free_range(struct dnode *dn, uint64_t start, uint64_t end,
301     struct dmu_tx *);

303 void dbuf_new_size(dmu_buf_impl_t *db, int size, dmu_tx_t *tx);

305 #define DB_DNODE(_db)           ((_db)->db_dnode_handle->dnh_dnode)
306 #define DB_DNODE_LOCK(_db)      ((_db)->db_dnode_handle->dnh_zrlock)
307 #define DB_DNODE_ENTER(_db)     (zrl_add(&DB_DNODE_LOCK(_db)))
308 #define DB_DNODE_EXIT(_db)      (zrl_remove(&DB_DNODE_LOCK(_db)))
309 #define DB_DNODE_HELD(_db)      (!zrl_is_zero(&DB_DNODE_LOCK(_db)))

311 void dbuf_init(void);
312 void dbuf_fini(void);

314 boolean_t dbuf_is_metadata(dmu_buf_impl_t *db);

316 #define DBUF_GET_BUFC_TYPE(_db) \
317         (dbuf_is_metadata(_db) ? ARC_BUFC_METADATA : ARC_BUFC_DATA)

319 #define DBUF_IS_CACHEABLE(_db)                                          \
320         ((_db)->db_objset->os_primary_cache == ZFS_CACHE_ALL ||         \
321         (dbuf_is_metadata(_db) &&                                       \
322         ((_db)->db_objset->os_primary_cache == ZFS_CACHE_METADATA)))

324 #define DBUF_IS_L2CACHEABLE(_db)                                        \
325         ((_db)->db_objset->os_secondary_cache == ZFS_CACHE_ALL ||       \
326         (dbuf_is_metadata(_db) &&                                       \
327         ((_db)->db_objset->os_secondary_cache == ZFS_CACHE_METADATA)))

329 #define DBUF_IS_L2COMPRESSIBLE(_db)                                     \
330         ((_db)->db_objset->os_compress != ZIO_COMPRESS_OFF ||           \
331         (dbuf_is_metadata(_db) && zfs_mdcomp_disable == B_FALSE))

333 #ifdef ZFS_DEBUG

335 /*
336  * There should be a ## between the string literal and fmt, to make it
337  * clear that we're joining two strings together, but gcc does not
338  * support that preprocessor token.
339  */
340 #define dprintf_dbuf(dbuf, fmt, ...) do { \
341         if (zfs_flags & ZFS_DEBUG_DPRINTF) { \
342         char __db_buf[32]; \
343         uint64_t __db_obj = (dbuf)->db.db_object; \
344         if (__db_obj == DMU_META_DNODE_OBJECT) \
345                 (void) strcpy(__db_buf, "mdn"); \
346         else \
347                 (void) snprintf(__db_buf, sizeof (__db_buf), "%lld", \
348                     (u_longlong_t)__db_obj); \
349         dprintf_ds((dbuf)->db_objset->os_dsl_dataset, \
350             "obj=%s lvl=%u blkid=%lld " fmt, \
351             __db_buf, (dbuf)->db_level, \
352             (u_longlong_t)(dbuf)->db_blkid, __VA_ARGS__); \
353         } \
354 _NOTE(CONSTCOND) } while (0)

356 #define dprintf_dbuf_bp(db, bp, fmt, ...) do {                  \
```

```
 357         if (zfs_flags & ZFS_DEBUG_DPRINTF) {                    \
 358         char *__blkbuf = kmem_alloc(BP_SPRINTF_LEN, KM_SLEEP);  \
 359         snprintf_blkptr(__blkbuf, BP_SPRINTF_LEN, bp);          \
 360         dprintf_dbuf(db, fmt " %s\n", __VA_ARGS__, __blkbuf);   \
 361         kmem_free(__blkbuf, BP_SPRINTF_LEN);                    \
 362         }                                                       \
 363 _NOTE(CONSTCOND) } while (0)

 365 #define DBUF_VERIFY(db) dbuf_verify(db)

 367 #else

 369 #define dprintf_dbuf(db, fmt, ...)
 370 #define dprintf_dbuf_bp(db, bp, fmt, ...)
 371 #define DBUF_VERIFY(db)

 373 #endif


 376 #ifdef  __cplusplus
 377 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   29810 Tue Oct 28 11:57:19 2014
new/usr/src/uts/common/fs/zfs/sys/dmu.h
Possibility to physically reserve space without writing leaf blocks
**********************************************************
______unchanged_portion_omitted_

 294 typedef void dmu_buf_evict_func_t(struct dmu_buf *db, void *user_ptr);

 296 /*
 297  * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
 298  */
 299 #define DMU_POOL_DIRECTORY_OBJECT       1
 300 #define DMU_POOL_CONFIG                 "config"
 301 #define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
 302 #define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
 303 #define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
 304 #define DMU_POOL_FEATURE_ENABLED_TXG    "feature_enabled_txg"
 305 #define DMU_POOL_ROOT_DATASET           "root_dataset"
 306 #define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
 307 #define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
 308 #define DMU_POOL_ERRLOG_LAST            "errlog_last"
 309 #define DMU_POOL_SPARES                 "spares"
 310 #define DMU_POOL_DEFLATE                "deflate"
 311 #define DMU_POOL_HISTORY                "history"
 312 #define DMU_POOL_PROPS                  "pool_props"
 313 #define DMU_POOL_L2CACHE                "l2cache"
 314 #define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
 315 #define DMU_POOL_DDT                    "DDT-%s-%s-%s"
 316 #define DMU_POOL_DDT_STATS              "DDT-statistics"
 317 #define DMU_POOL_CREATION_VERSION       "creation_version"
 318 #define DMU_POOL_SCAN                   "scan"
 319 #define DMU_POOL_FREE_BPOBJ             "free_bpobj"
 320 #define DMU_POOL_BPTREE_OBJ             "bptree_obj"
 321 #define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"

 323 /*
 324  * Allocate an object from this objset.  The range of object numbers
 325  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
 326  *
 327  * The transaction must be assigned to a txg.  The newly allocated
 328  * object will be "held" in the transaction (ie. you can modify the
 329  * newly allocated object in this transaction).
 330  *
 331  * dmu_object_alloc() chooses an object and returns it in *objectp.
 332  *
 333  * dmu_object_claim() allocates a specific object number.  If that
 334  * number is already allocated, it fails and returns EEXIST.
 335  *
 336  * Return 0 on success, or ENOSPC or EEXIST as specified above.
 337  */
 338 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
 339     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 340 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 341     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 342 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 343     int blocksize, dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *txp);

 345 /*
 346  * Free an object from this objset.
 347  *
 348  * The object's data will be freed as well (ie. you don't need to call
 349  * dmu_free(object, 0, -1, tx)).
 350  *
 351  * The object need not be held in the transaction.
 352  *
```



```
 353  * If there are any holds on this object's buffers (via dmu_buf_hold()),
 354  * or tx holds on the object (via dmu_tx_hold_object()), you can not
 355  * free it; it fails and returns EBUSY.
 356  *
 357  * If the object is not allocated, it fails and returns ENOENT.
 358  *
 359  * Return 0 on success, or EBUSY or ENOENT as specified above.
 360  */
 361 int dmu_object_free(objset_t *os, uint64_t object, dmu_tx_t *tx);

 363 /*
 364  * Find the next allocated or free object.
 365  *
 366  * The objectp parameter is in-out.  It will be updated to be the next
 367  * object which is allocated.  Ignore objects which have not been
 368  * modified since txg.
 369  *
 370  * XXX Can only be called on a objset with no dirty data.
 371  *
 372  * Returns 0 on success, or ENOENT if there are no more objects.
 373  */
 374 int dmu_object_next(objset_t *os, uint64_t *objectp,
 375     boolean_t hole, uint64_t txg);

 377 /*
 378  * Set the data blocksize for an object.
 379  *
 380  * The object cannot have any blocks allcated beyond the first.  If
 381  * the first block is allocated already, the new size must be greater
 382  * than the current block size.  If these conditions are not met,
 383  * ENOTSUP will be returned.
 384  *
 385  * Returns 0 on success, or EBUSY if there are any holds on the object
 386  * contents, or ENOTSUP as described above.
 387  */
 388 int dmu_object_set_blocksize(objset_t *os, uint64_t object, uint64_t size,
 389     int ibs, dmu_tx_t *tx);

 391 /*
 392  * Set the checksum property on a dnode.  The new checksum algorithm will
 393  * apply to all newly written blocks; existing blocks will not be affected.
 394  */
 395 void dmu_object_set_checksum(objset_t *os, uint64_t object, uint8_t checksum,
 396     dmu_tx_t *tx);

 398 /*
 399  * Set the compress property on a dnode.  The new compression algorithm will
 400  * apply to all newly written blocks; existing blocks will not be affected.
 401  */
 402 void dmu_object_set_compress(objset_t *os, uint64_t object, uint8_t compress,
 403     dmu_tx_t *tx);

 405 void
 406 dmu_write_embedded(objset_t *os, uint64_t object, uint64_t offset,
 407     void *data, uint8_t etype, uint8_t comp, int uncompressed_size,
 408     int compressed_size, int byteorder, dmu_tx_t *tx);

 410 /*
 411  * Decide how to write a block: checksum, compression, number of copies, etc.
 412  */
 413 #define WP_NOFILL       0x1
 414 #define WP_DMU_SYNC     0x2
 415 #define WP_SPILL        0x4

 417 void dmu_write_policy(objset_t *os, struct dnode *dn, int level, int wp,
 418     struct zio_prop *zp);
```

```
419 /*
420  * The bonus data is accessed more or less like a regular buffer.
421  * You must dmu_bonus_hold() to get the buffer, which will give you a
422  * dmu_buf_t with db_offset==-1ULL, and db_size = the size of the bonus
423  * data.  As with any normal buffer, you must call dmu_buf_read() to
424  * read db_data, dmu_buf_will_dirty() before modifying it, and the
425  * object must be held in an assigned transaction before calling
426  * dmu_buf_will_dirty.  You may use dmu_buf_set_user() on the bonus
427  * buffer as well.  You must release your hold with dmu_buf_rele().
428  *
429  * Returns ENOENT, EIO, or 0.
430  */
431 int dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **);
432 int dmu_bonus_max(void);
433 int dmu_set_bonus(dmu_buf_t *, int, dmu_tx_t *);
434 int dmu_set_bonustype(dmu_buf_t *, dmu_object_type_t, dmu_tx_t *);
435 dmu_object_type_t dmu_get_bonustype(dmu_buf_t *);
436 int dmu_rm_spill(objset_t *, uint64_t, dmu_tx_t *);

438 /*
439  * Special spill buffer support used by "SA" framework
440  */

442 int dmu_spill_hold_by_bonus(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);
443 int dmu_spill_hold_by_dnode(struct dnode *dn, uint32_t flags,
444     void *tag, dmu_buf_t **dbp);
445 int dmu_spill_hold_existing(dmu_buf_t *bonus, void *tag, dmu_buf_t **dbp);

447 /*
448  * Obtain the DMU buffer from the specified object which contains the
449  * specified offset.  dmu_buf_hold() puts a "hold" on the buffer, so
450  * that it will remain in memory.  You must release the hold with
451  * dmu_buf_rele().  You musn't access the dmu_buf_t after releasing your
452  * hold.  You must have a hold on any dmu_buf_t* you pass to the DMU.
453  *
454  * You must call dmu_buf_read, dmu_buf_will_dirty, or dmu_buf_will_fill
455  * on the returned buffer before reading or writing the buffer's
456  * db_data.  The comments for those routines describe what particular
457  * operations are valid after calling them.
458  *
459  * The object number must be a valid, allocated object number.
460  */
461 int dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
462     void *tag, dmu_buf_t **, int flags);
463 void dmu_buf_add_ref(dmu_buf_t *db, void* tag);
464 void dmu_buf_rele(dmu_buf_t *db, void *tag);
465 uint64_t dmu_buf_refcount(dmu_buf_t *db);

467 /*
468  * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
469  * range of an object.  A pointer to an array of dmu_buf_t*'s is
470  * returned (in *dbpp).
471  *
472  * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*'s, and
473  * frees the array.  The hold on the array of buffers MUST be released
474  * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
475  * individually with dmu_buf_rele.
476  */
477 int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
478     uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
479 void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

481 /*
482  * Returns NULL on success, or the existing user ptr if it's already
483  * been set.
484  *
```



```
485  * user_ptr is for use by the user and can be obtained via dmu_buf_get_user().
486  *
487  * user_data_ptr_ptr should be NULL, or a pointer to a pointer which
488  * will be set to db->db_data when you are allowed to access it.  Note
489  * that db->db_data (the pointer) can change when you do dmu_buf_read(),
490  * dmu_buf_tryupgrade(), dmu_buf_will_dirty(), or dmu_buf_will_fill().
491  * *user_data_ptr_ptr will be set to the new value when it changes.
492  *
493  * If non-NULL, pageout func will be called when this buffer is being
494  * excised from the cache, so that you can clean up the data structure
495  * pointed to by user_ptr.
496  *
497  * dmu_evict_user() will call the pageout func for all buffers in a
498  * objset with a given pageout func.
499  */
500 void *dmu_buf_set_user(dmu_buf_t *db, void *user_ptr, void *user_data_ptr_ptr,
501     dmu_buf_evict_func_t *pageout_func);
502 /*
503  * set_user_ie is the same as set_user, but request immediate eviction
504  * when hold count goes to zero.
505  */
506 void *dmu_buf_set_user_ie(dmu_buf_t *db, void *user_ptr,
507     void *user_data_ptr_ptr, dmu_buf_evict_func_t *pageout_func);
508 void *dmu_buf_update_user(dmu_buf_t *db_fake, void *old_user_ptr,
509     void *user_ptr, void *user_data_ptr_ptr,
510     dmu_buf_evict_func_t *pageout_func);
511 void dmu_evict_user(objset_t *os, dmu_buf_evict_func_t *func);

513 /*
514  * Returns the user_ptr set with dmu_buf_set_user(), or NULL if not set.
515  */
516 void *dmu_buf_get_user(dmu_buf_t *db);

518 /*
519  * Returns the blkptr associated with this dbuf, or NULL if not set.
520  */
521 struct blkptr *dmu_buf_get_blkptr(dmu_buf_t *db);

523 /*
524  * Indicate that you are going to modify the buffer's data (db_data).
525  *
526  * The transaction (tx) must be assigned to a txg (ie. you've called
527  * dmu_tx_assign()).  The buffer's object must be held in the tx
528  * (ie. you've called dmu_tx_hold_object(tx, db->db_object)).
529  */
530 void dmu_buf_will_dirty(dmu_buf_t *db, dmu_tx_t *tx);

532 /*
533  * Tells if the given dbuf is freeable.
534  */
535 boolean_t dmu_buf_freeable(dmu_buf_t *);

537 /*
538  * You must create a transaction, then hold the objects which you will
539  * (or might) modify as part of this transaction.  Then you must assign
540  * the transaction to a transaction group.  Once the transaction has
541  * been assigned, you can modify buffers which belong to held objects as
542  * part of this transaction.  You can't modify buffers before the
543  * transaction has been assigned; you can't modify buffers which don't
544  * belong to objects which this transaction holds; you can't hold
545  * objects once the transaction has been assigned.  You may hold an
546  * object which you are going to free (with dmu_object_free()), but you
547  * don't have to.
548  *
549  * You can abort the transaction before it has been assigned.
550  *
```

```
551  * Note that you may hold buffers (with dmu_buf_hold) at any time,
552  * regardless of transaction state.
553  */

555 #define DMU_NEW_OBJECT  (-1ULL)
556 #define DMU_OBJECT_END  (-1ULL)

558 dmu_tx_t *dmu_tx_create(objset_t *os);
559 void dmu_tx_hold_write(dmu_tx_t *tx, uint64_t object, uint64_t off, int len);
560 void dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off,
561     uint64_t len);
562 void dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name);
563 void dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object);
564 void dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object);
565 void dmu_tx_hold_sa(dmu_tx_t *tx, struct sa_handle *hdl, boolean_t may_grow);
566 void dmu_tx_hold_sa_create(dmu_tx_t *tx, int total_size);
567 void dmu_tx_abort(dmu_tx_t *tx);
568 int dmu_tx_assign(dmu_tx_t *tx, enum txg_how txg_how);
569 void dmu_tx_wait(dmu_tx_t *tx);
570 void dmu_tx_commit(dmu_tx_t *tx);
571 void dmu_tx_mark_netfree(dmu_tx_t *tx);

573 /*
574  * To register a commit callback, dmu_tx_callback_register() must be called.
575  *
576  * dcb_data is a pointer to caller private data that is passed on as a
577  * callback parameter. The caller is responsible for properly allocating and
578  * freeing it.
579  *
580  * When registering a callback, the transaction must be already created, but
581  * it cannot be committed or aborted. It can be assigned to a txg or not.
582  *
583  * The callback will be called after the transaction has been safely written
584  * to stable storage and will also be called if the dmu_tx is aborted.
585  * If there is any error which prevents the transaction from being committed to
586  * disk, the callback will be called with a value of error != 0.
587  */
588 typedef void dmu_tx_callback_func_t(void *dcb_data, int error);

590 void dmu_tx_callback_register(dmu_tx_t *tx, dmu_tx_callback_func_t *dcb_func,
591     void *dcb_data);

593 /*
594  * Free up the data blocks for a defined range of a file.  If size is
595  * -1, the range from offset to end-of-file is freed.
596  */
597 int dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
598         uint64_t size, dmu_tx_t *tx);
599 int dmu_free_long_range(objset_t *os, uint64_t object, uint64_t offset,
600         uint64_t size);
601 int dmu_free_long_object(objset_t *os, uint64_t object);

603 /*
604  * Convenience functions.
605  *
606  * Canfail routines will return 0 on success, or an errno if there is a
607  * nonrecoverable I/O error.
608  */
609 #define DMU_READ_PREFETCH       0 /* prefetch */
610 #define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
611 int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
612         void *buf, uint32_t flags);
613 void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
614         const void *buf, dmu_tx_t *tx);
615 void dmu_write_zero(objset_t *os, uint64_t object, uint64_t offset, uint64_t siz
616 #endif /* ! codereview */
```



```
617 void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
618         dmu_tx_t *tx);
619 int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
620 int dmu_read_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size);
621 int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
622     dmu_tx_t *tx);
623 int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
624     dmu_tx_t *tx);
625 int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
626     uint64_t size, struct page *pp, dmu_tx_t *tx);
627 struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
628 void dmu_return_arcbuf(struct arc_buf *buf);
629 void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
630     dmu_tx_t *tx);
631 int dmu_xuio_init(struct xuio *uio, int niov);
632 void dmu_xuio_fini(struct xuio *uio);
633 int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
634     size_t n);
635 int dmu_xuio_cnt(struct xuio *uio);
636 struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
637 void dmu_xuio_clear(struct xuio *uio, int i);
638 void xuio_stat_wbuf_copied();
639 void xuio_stat_wbuf_nocopy();

641 extern int zfs_prefetch_disable;

643 /*
644  * Asynchronously try to read in the data.
645  */
646 void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
647     uint64_t len);

649 typedef struct dmu_object_info {
650         /* All sizes are in bytes unless otherwise indicated. */
651         uint32_t doi_data_block_size;
652         uint32_t doi_metadata_block_size;
653         dmu_object_type_t doi_type;
654         dmu_object_type_t doi_bonus_type;
655         uint64_t doi_bonus_size;
656         uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
657         uint8_t doi_checksum;
658         uint8_t doi_compress;
659         uint8_t doi_nblkptr;
660         uint8_t doi_pad[4];
661         uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
662         uint64_t doi_max_offset;
663         uint64_t doi_fill_count;                /* number of non-empty blocks */
664 } dmu_object_info_t;

666 typedef void arc_byteswap_func_t(void *buf, size_t size);

668 typedef struct dmu_object_type_info {
669         dmu_object_byteswap_t   ot_byteswap;
670         boolean_t               ot_metadata;
671         char                    *ot_name;
672 } dmu_object_type_info_t;

674 typedef struct dmu_object_byteswap_info {
675         arc_byteswap_func_t     *ob_func;
676         char                    *ob_name;
677 } dmu_object_byteswap_info_t;

679 extern const dmu_object_type_info_t dmu_ot[DMU_OT_NUMTYPES];
680 extern const dmu_object_byteswap_info_t dmu_ot_byteswap[DMU_BSWAP_NUMFUNCS];

682 /*
```

```
 * Get information on a DMU object.
 *
 * Return 0 on success or ENOENT if object is not allocated.
 *
 * If doi is NULL, just indicates whether the object exists.
 */
int dmu_object_info(objset_t *os, uint64_t object, dmu_object_info_t *doi);
/* Like dmu_object_info, but faster if you have a held dnode in hand. */
void dmu_object_info_from_dnode(struct dnode *dn, dmu_object_info_t *doi);
/* Like dmu_object_info, but faster if you have a held dbuf in hand. */
void dmu_object_info_from_db(dmu_buf_t *db, dmu_object_info_t *doi);
/*
 * Like dmu_object_info_from_db, but faster still when you only care about
 * the size.  This is specifically optimized for zfs_getattr().
 */
void dmu_object_size_from_db(dmu_buf_t *db, uint32_t *blksize,
    u_longlong_t *nblk512);

typedef struct dmu_objset_stats {
        uint64_t dds_num_clones; /* number of clones of this */
        uint64_t dds_creation_txg;
        uint64_t dds_guid;
        dmu_objset_type_t dds_type;
        uint8_t dds_is_snapshot;
        uint8_t dds_inconsistent;
        char dds_origin[MAXNAMELEN];
} dmu_objset_stats_t;

/*
 * Get stats on a dataset.
 */
void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);

/*
 * Add entries to the nvlist for all the objset's properties.  See
 * zfs_prop_table[] and zfs(1m) for details on the properties.
 */
void dmu_objset_stats(objset_t *os, struct nvlist *nv);

/*
 * Get the space usage statistics for statvfs().
 *
 * refdbytes is the amount of space "referenced" by this objset.
 * availbytes is the amount of space available to this objset, taking
 * into account quotas & reservations, assuming that no other objsets
 * use the space first.  These values correspond to the 'referenced' and
 * 'available' properties, described in the zfs(1m) manpage.
 *
 * usedobjs and availobjs are the number of objects currently allocated,
 * and available.
 */
void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp);

/*
 * The fsid_guid is a 56-bit ID that can change to avoid collisions.
 * (Contrast with the ds_guid which is a 64-bit ID that will never
 * change, so there is a small probability that it will collide.)
 */
uint64_t dmu_objset_fsid_guid(objset_t *os);

/*
 * Get the [cm]time for an objset's snapshot dir
 */
timestruc_t dmu_objset_snap_cmtime(objset_t *os);
```



```
int dmu_objset_is_snapshot(objset_t *os);

extern struct spa *dmu_objset_spa(objset_t *os);
extern struct zilog *dmu_objset_zil(objset_t *os);
extern struct dsl_pool *dmu_objset_pool(objset_t *os);
extern struct dsl_dataset *dmu_objset_ds(objset_t *os);
extern void dmu_objset_name(objset_t *os, char *buf);
extern dmu_objset_type_t dmu_objset_type(objset_t *os);
extern uint64_t dmu_objset_id(objset_t *os);
extern zfs_sync_type_t dmu_objset_syncprop(objset_t *os);
extern zfs_logbias_op_t dmu_objset_logbias(objset_t *os);
extern int dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
    uint64_t *id, uint64_t *offp, boolean_t *case_conflict);
extern int dmu_snapshot_realname(objset_t *os, char *name, char *real,
    int maxlen, boolean_t *conflict);
extern int dmu_dir_list_next(objset_t *os, int namelen, char *name,
    uint64_t *idp, uint64_t *offp);

typedef int objset_used_cb_t(dmu_object_type_t bonustype,
    void *bonus, uint64_t *userp, uint64_t *groupp);
extern void dmu_objset_register_type(dmu_objset_type_t ost,
    objset_used_cb_t *cb);
extern void dmu_objset_set_user(objset_t *os, void *user_ptr);
extern void *dmu_objset_get_user(objset_t *os);

/*
 * Return the txg number for the given assigned transaction.
 */
uint64_t dmu_tx_get_txg(dmu_tx_t *tx);

/*
 * Synchronous write.
 * If a parent zio is provided this function initiates a write on the
 * provided buffer as a child of the parent zio.
 * In the absence of a parent zio, the write is completed synchronously.
 * At write completion, blk is filled with the bp of the written block.
 * Note that while the data covered by this function will be on stable
 * storage when the write completes this new data does not become a
 * permanent part of the file until the associated transaction commits.
 */

/*
 * {zfs,zvol,ztest}_get_done() args
 */
typedef struct zgd {
        struct zilog    *zgd_zilog;
        struct blkptr   *zgd_bp;
        dmu_buf_t       *zgd_db;
        struct rl       *zgd_rl;
        void            *zgd_private;
} zgd_t;

typedef void dmu_sync_cb_t(zgd_t *arg, int error);
int dmu_sync(struct zio *zio, uint64_t txg, dmu_sync_cb_t *done, zgd_t *zgd);

/*
 * Find the next hole or data block in file starting at *off
 * Return found offset in *off. Return ESRCH for end of file.
 */
int dmu_offset_next(objset_t *os, uint64_t object, boolean_t hole,
    uint64_t *off);

/*
 * Initial setup and final teardown.
 */
extern void dmu_init(void);
```

```
815 extern void dmu_fini(void);

817 typedef void (*dmu_traverse_cb_t)(objset_t *os, void *arg, struct blkptr *bp,
818     uint64_t object, uint64_t offset, int len);
819 void dmu_traverse_objset(objset_t *os, uint64_t txg_start,
820     dmu_traverse_cb_t cb, void *arg);

822 int dmu_diff(const char *tosnap_name, const char *fromsnap_name,
823     struct vnode *vp, offset_t *offp);

825 /* CRC64 table */
826 #define ZFS_CRC64_POLY  0xC96C5795D7870F42ULL   /* ECMA-182, reflected form */
827 extern uint64_t zfs_crc64_table[256];

829 extern int zfs_mdcomp_disable;

831 #ifdef  __cplusplus
832 }
833 #endif

835 #endif  /* _SYS_DMU_H */
```

****
32347 Tue Oct 28 11:57:19 2014
new/usr/src/uts/common/fs/zfs/sys/spa.h
Possibility to physically reserve space without writing leaf blocks
****
_____unchanged_portion_omitted_

```
316 /*
317  * Macros to get and set fields in a bp or DVA.
318  */
319 #define DVA_GET_ASIZE(dva)      \
320         BF64_GET_SB((dva)->dva_word[0], 0, SPA_ASIZEBITS, SPA_MINBLOCKSHIFT, 0)
321 #define DVA_SET_ASIZE(dva, x)   \
322         BF64_SET_SB((dva)->dva_word[0], 0, SPA_ASIZEBITS, \
323         SPA_MINBLOCKSHIFT, 0, x)

325 #define DVA_GET_GRID(dva)       BF64_GET((dva)->dva_word[0], 24, 8)
326 #define DVA_SET_GRID(dva, x)    BF64_SET((dva)->dva_word[0], 24, 8, x)

328 #define DVA_GET_VDEV(dva)       BF64_GET((dva)->dva_word[0], 32, 32)
329 #define DVA_SET_VDEV(dva, x)    BF64_SET((dva)->dva_word[0], 32, 32, x)

331 #define DVA_GET_OFFSET(dva)     \
332         BF64_GET_SB((dva)->dva_word[1], 0, 63, SPA_MINBLOCKSHIFT, 0)
333 #define DVA_SET_OFFSET(dva, x)  \
334         BF64_SET_SB((dva)->dva_word[1], 0, 63, SPA_MINBLOCKSHIFT, 0, x)

336 #define DVA_GET_GANG(dva)       BF64_GET((dva)->dva_word[1], 63, 1)
337 #define DVA_SET_GANG(dva, x)    BF64_SET((dva)->dva_word[1], 63, 1, x)

339 #define BP_GET_LSIZE(bp)        \
340         (BP_IS_EMBEDDED(bp) ?   \
341         (BPE_GET_ETYPE(bp) == BP_EMBEDDED_TYPE_DATA ? BPE_GET_LSIZE(bp) : 0): \
342         BF64_GET_SB((bp)->blk_prop, 0, SPA_LSIZEBITS, SPA_MINBLOCKSHIFT, 1))
343 #define BP_SET_LSIZE(bp, x)     do { \
344         ASSERT(!BP_IS_EMBEDDED(bp)); \
345         BF64_SET_SB((bp)->blk_prop, \
346             0, SPA_LSIZEBITS, SPA_MINBLOCKSHIFT, 1, x); \
347 _NOTE(CONSTCOND) } while (0)

349 #define BP_GET_PSIZE(bp)        \
350         (BP_IS_EMBEDDED(bp) ? 0 : \
351         BF64_GET_SB((bp)->blk_prop, 16, SPA_PSIZEBITS, SPA_MINBLOCKSHIFT, 1))
352 #define BP_SET_PSIZE(bp, x)     do { \
353         ASSERT(!BP_IS_EMBEDDED(bp)); \
354         BF64_SET_SB((bp)->blk_prop, \
355             16, SPA_PSIZEBITS, SPA_MINBLOCKSHIFT, 1, x); \
356 _NOTE(CONSTCOND) } while (0)

358 #define BP_GET_COMPRESS(bp)             BF64_GET((bp)->blk_prop, 32, 7)
359 #define BP_SET_COMPRESS(bp, x)          BF64_SET((bp)->blk_prop, 32, 7, x)

361 #define BP_IS_EMBEDDED(bp)              BF64_GET((bp)->blk_prop, 39, 1)
362 #define BP_SET_EMBEDDED(bp, x)          BF64_SET((bp)->blk_prop, 39, 1, x)

364 #define BP_GET_CHECKSUM(bp)             \
365         (BP_IS_EMBEDDED(bp) ? ZIO_CHECKSUM_OFF : \
366         BF64_GET((bp)->blk_prop, 40, 8))
367 #define BP_SET_CHECKSUM(bp, x)          do { \
368         ASSERT(!BP_IS_EMBEDDED(bp)); \
369         BF64_SET((bp)->blk_prop, 40, 8, x); \
370 _NOTE(CONSTCOND) } while (0)

372 #define BP_GET_TYPE(bp)                 BF64_GET((bp)->blk_prop, 48, 8)
373 #define BP_SET_TYPE(bp, x)              BF64_SET((bp)->blk_prop, 48, 8, x)
```



```
375 #define BP_GET_LEVEL(bp)                BF64_GET((bp)->blk_prop, 56, 5)
376 #define BP_SET_LEVEL(bp, x)             BF64_SET((bp)->blk_prop, 56, 5, x)

378 #define BP_GET_PROP_RESERVATION(bp)          BF64_GET((bp)->blk_prop, 61, 1)
379 #define BP_SET_PROP_RESERVATION(bp, x)       BF64_SET((bp)->blk_prop, 61, 1, x)

381 #endif /* ! codereview */
382 #define BP_GET_DEDUP(bp)                BF64_GET((bp)->blk_prop, 62, 1)
383 #define BP_SET_DEDUP(bp, x)             BF64_SET((bp)->blk_prop, 62, 1, x)

385 #define BP_GET_BYTEORDER(bp)            BF64_GET((bp)->blk_prop, 63, 1)
386 #define BP_SET_BYTEORDER(bp, x)         BF64_SET((bp)->blk_prop, 63, 1, x)

388 #define BP_PHYSICAL_BIRTH(bp)           \
389         (BP_IS_EMBEDDED(bp) ? 0 : \
390         (bp)->blk_phys_birth ? (bp)->blk_phys_birth : (bp)->blk_birth)

392 #define BP_SET_BIRTH(bp, logical, physical)     \
393 {                                               \
394         ASSERT(!BP_IS_EMBEDDED(bp));            \
395         (bp)->blk_birth = (logical);            \
396         (bp)->blk_phys_birth = ((logical) == (physical) ? 0 : (physical)); \
397 }

399 #define BP_GET_FILL(bp) (BP_IS_EMBEDDED(bp) ? 1 : (bp)->blk_fill)

401 #define BP_GET_ASIZE(bp)        \
402         (BP_IS_EMBEDDED(bp) ? 0 : \
403         DVA_GET_ASIZE(&(bp)->blk_dva[0]) + \
404         DVA_GET_ASIZE(&(bp)->blk_dva[1]) + \
405         DVA_GET_ASIZE(&(bp)->blk_dva[2]))

407 #define BP_GET_UCSIZE(bp) \
408         ((BP_GET_LEVEL(bp) > 0 || DMU_OT_IS_METADATA(BP_GET_TYPE(bp))) ? \
409         BP_GET_PSIZE(bp) : BP_GET_LSIZE(bp))

411 #define BP_GET_NDVAS(bp)        \
412         (BP_IS_EMBEDDED(bp) ? 0 : \
413         !!DVA_GET_ASIZE(&(bp)->blk_dva[0]) + \
414         !!DVA_GET_ASIZE(&(bp)->blk_dva[1]) + \
415         !!DVA_GET_ASIZE(&(bp)->blk_dva[2]))

417 #define BP_COUNT_GANG(bp)       \
418         (BP_IS_EMBEDDED(bp) ? 0 : \
419         (DVA_GET_GANG(&(bp)->blk_dva[0]) + \
420         DVA_GET_GANG(&(bp)->blk_dva[1]) + \
421         DVA_GET_GANG(&(bp)->blk_dva[2])))

423 #define DVA_EQUAL(dva1, dva2)   \
424         ((dva1)->dva_word[1] == (dva2)->dva_word[1] && \
425         (dva1)->dva_word[0] == (dva2)->dva_word[0])

427 #define BP_EQUAL(bp1, bp2)      \
428         (BP_PHYSICAL_BIRTH(bp1) == BP_PHYSICAL_BIRTH(bp2) &&    \
429         (bp1)->blk_birth == (bp2)->blk_birth &&                 \
430         DVA_EQUAL(&(bp1)->blk_dva[0], &(bp2)->blk_dva[0]) &&    \
431         DVA_EQUAL(&(bp1)->blk_dva[1], &(bp2)->blk_dva[1]) &&    \
432         DVA_EQUAL(&(bp1)->blk_dva[2], &(bp2)->blk_dva[2]))

434 #define ZIO_CHECKSUM_EQUAL(zc1, zc2) \
435         (0 == (((zc1).zc_word[0] - (zc2).zc_word[0]) | \
436         ((zc1).zc_word[1] - (zc2).zc_word[1]) | \
437         ((zc1).zc_word[2] - (zc2).zc_word[2]) | \
438         ((zc1).zc_word[3] - (zc2).zc_word[3])))

440 #define DVA_IS_VALID(dva)       (DVA_GET_ASIZE(dva) != 0)
```

```
#define ZIO_SET_CHECKSUM(zcp, w0, w1, w2, w3)   \
{                                               \
        (zcp)->zc_word[0] = w0;                 \
        (zcp)->zc_word[1] = w1;                 \
        (zcp)->zc_word[2] = w2;                 \
        (zcp)->zc_word[3] = w3;                 \
}

#define BP_IDENTITY(bp)         (ASSERT(!BP_IS_EMBEDDED(bp)), &(bp)->blk_dva[0])
#define BP_IS_GANG(bp)          \
        (BP_IS_EMBEDDED(bp) ? B_FALSE : DVA_GET_GANG(BP_IDENTITY(bp)))
#define DVA_IS_EMPTY(dva)       ((dva)->dva_word[0] == 0ULL &&  \
                                 (dva)->dva_word[1] == 0ULL)
#define BP_IS_HOLE(bp) \
        (!BP_IS_EMBEDDED(bp) && DVA_IS_EMPTY(BP_IDENTITY(bp)))

/* BP_IS_RAIDZ(bp) assumes no block compression */
#define BP_IS_RAIDZ(bp)         (DVA_GET_ASIZE(&(bp)->blk_dva[0]) > \
                                 BP_GET_PSIZE(bp))

#define BP_ZERO(bp)                             \
{                                               \
        (bp)->blk_dva[0].dva_word[0] = 0;       \
        (bp)->blk_dva[0].dva_word[1] = 0;       \
        (bp)->blk_dva[1].dva_word[0] = 0;       \
        (bp)->blk_dva[1].dva_word[1] = 0;       \
        (bp)->blk_dva[2].dva_word[0] = 0;       \
        (bp)->blk_dva[2].dva_word[1] = 0;       \
        (bp)->blk_prop = 0;                     \
        (bp)->blk_pad[0] = 0;                   \
        (bp)->blk_pad[1] = 0;                   \
        (bp)->blk_phys_birth = 0;               \
        (bp)->blk_birth = 0;                    \
        (bp)->blk_fill = 0;                     \
        ZIO_SET_CHECKSUM(&(bp)->blk_cksum, 0, 0, 0, 0); \
}

#ifdef _BIG_ENDIAN
#define ZFS_HOST_BYTEORDER      (0ULL)
#else
#define ZFS_HOST_BYTEORDER      (1ULL)
#endif

#define BP_SHOULD_BYTESWAP(bp)  (BP_GET_BYTEORDER(bp) != ZFS_HOST_BYTEORDER)

#define BP_SPRINTF_LEN  320

/*
 * This macro allows code sharing between zfs, libzpool, and mdb.
 * 'func' is either snprintf() or mdb_snprintf().
 * 'ws' (whitespace) can be ' ' for single-line format, '\n' for multi-line.
 */
#define SNPRINTF_BLKPTR(func, ws, buf, size, bp, type, checksum, compress) \
{                                                                       \
        static const char *copyname[] =                                 \
            { "zero", "single", "double", "triple" };                   \
        int len = 0;                                                    \
        int copies = 0;                                                 \
                                                                        \
        if (bp == NULL) {                                               \
                len += func(buf + len, size - len, "<NULL>");           \
        } else if (BP_IS_HOLE(bp)) {                                    \
                len += func(buf + len, size - len, "<hole>");           \
                if (bp->blk_birth > 0) {                                \
                        len += func(buf + len, size - len,              \
```



```
                            " birth=%lluL",                             \
                            (u_longlong_t)bp->blk_birth);               \
                }                                                       \
        } else if (BP_IS_EMBEDDED(bp)) {                                \
                len = func(buf + len, size - len,                       \
                    "EMBEDDED [L%llu %s] et=%u %s "                     \
                    "size=%llxL/%llxP birth=%lluL",                     \
                    (u_longlong_t)BP_GET_LEVEL(bp),                     \
                    type,                                               \
                    (int)BPE_GET_ETYPE(bp),                             \
                    compress,                                           \
                    (u_longlong_t)BPE_GET_LSIZE(bp),                    \
                    (u_longlong_t)BPE_GET_PSIZE(bp),                    \
                    (u_longlong_t)bp->blk_birth);                       \
        } else {                                                        \
                for (int d = 0; d < BP_GET_NDVAS(bp); d++) {            \
                        const dva_t *dva = &bp->blk_dva[d];             \
                        if (DVA_IS_VALID(dva))                          \
                                copies++;                               \
                        len += func(buf + len, size - len,              \
                            "DVA[%d]=<%llu:%llx:%llx>%c", d,            \
                            (u_longlong_t)DVA_GET_VDEV(dva),            \
                            (u_longlong_t)DVA_GET_OFFSET(dva),          \
                            (u_longlong_t)DVA_GET_ASIZE(dva),           \
                            ws);                                        \
                }                                                       \
                if (BP_IS_GANG(bp) &&                                   \
                    DVA_GET_ASIZE(&bp->blk_dva[2]) <=                   \
                    DVA_GET_ASIZE(&bp->blk_dva[1]) / 2)                 \
                        copies--;                                       \
                len += func(buf + len, size - len,                      \
                    "[L%llu %s] %s %s %s %s %s %s%c"                    \
                    "size=%llxL/%llxP birth=%lluL/%lluP fill=%llu%c"    \
                    "cksum=%llx:%llx:%llx:%llx",                        \
                    (u_longlong_t)BP_GET_LEVEL(bp),                     \
                    type,                                               \
                    checksum,                                           \
                    compress,                                           \
                    BP_GET_BYTEORDER(bp) == 0 ? "BE" : "LE",            \
                    BP_IS_GANG(bp) ? "gang" : "contiguous",             \
                    BP_GET_DEDUP(bp) ? "dedup" : "unique",              \
                    copyname[copies],                                   \
                    ws,                                                 \
                    (u_longlong_t)BP_GET_LSIZE(bp),                     \
                    (u_longlong_t)BP_GET_PSIZE(bp),                     \
                    (u_longlong_t)bp->blk_birth,                        \
                    (u_longlong_t)BP_PHYSICAL_BIRTH(bp),                \
                    (u_longlong_t)BP_GET_FILL(bp),                      \
                    ws,                                                 \
                    (u_longlong_t)bp->blk_cksum.zc_word[0],             \
                    (u_longlong_t)bp->blk_cksum.zc_word[1],             \
                    (u_longlong_t)bp->blk_cksum.zc_word[2],             \
                    (u_longlong_t)bp->blk_cksum.zc_word[3]);            \
        }                                                               \
        ASSERT(len < size);                                             \
}

#include <sys/dmu.h>

#define BP_GET_BUFC_TYPE(bp)                                            \
        (((BP_GET_LEVEL(bp) > 0) || (DMU_OT_IS_METADATA(BP_GET_TYPE(bp)))) ? \
        ARC_BUFC_METADATA : ARC_BUFC_DATA)

typedef enum spa_import_type {
        SPA_IMPORT_EXISTING,
        SPA_IMPORT_ASSEMBLE
```

```
573 } spa_import_type_t;

575 /* state manipulation functions */
576 extern int spa_open(const char *pool, spa_t **, void *tag);
577 extern int spa_open_rewind(const char *pool, spa_t **, void *tag,
578     nvlist_t *policy, nvlist_t **config);
579 extern int spa_get_stats(const char *pool, nvlist_t **config, char *altroot,
580     size_t buflen);
581 extern int spa_create(const char *pool, nvlist_t *config, nvlist_t *props,
582     nvlist_t *zplprops);
583 extern int spa_import_rootpool(char *devpath, char *devid);
584 extern int spa_import(const char *pool, nvlist_t *config, nvlist_t *props,
585     uint64_t flags);
586 extern nvlist_t *spa_tryimport(nvlist_t *tryconfig);
587 extern int spa_destroy(char *pool);
588 extern int spa_export(char *pool, nvlist_t **oldconfig, boolean_t force,
589     boolean_t hardforce);
590 extern int spa_reset(char *pool);
591 extern void spa_async_request(spa_t *spa, int flag);
592 extern void spa_async_unrequest(spa_t *spa, int flag);
593 extern void spa_async_suspend(spa_t *spa);
594 extern void spa_async_resume(spa_t *spa);
595 extern spa_t *spa_inject_addref(char *pool);
596 extern void spa_inject_delref(spa_t *spa);
597 extern void spa_scan_stat_init(spa_t *spa);
598 extern int spa_scan_get_stats(spa_t *spa, pool_scan_stat_t *ps);

600 #define SPA_ASYNC_CONFIG_UPDATE 0x01
601 #define SPA_ASYNC_REMOVE        0x02
602 #define SPA_ASYNC_PROBE         0x04
603 #define SPA_ASYNC_RESILVER_DONE 0x08
604 #define SPA_ASYNC_RESILVER      0x10
605 #define SPA_ASYNC_AUTOEXPAND    0x20
606 #define SPA_ASYNC_REMOVE_DONE   0x40
607 #define SPA_ASYNC_REMOVE_STOP   0x80

609 /*
610  * Controls the behavior of spa_vdev_remove().
611  */
612 #define SPA_REMOVE_UNSPARE      0x01
613 #define SPA_REMOVE_DONE         0x02

615 /* device manipulation */
616 extern int spa_vdev_add(spa_t *spa, nvlist_t *nvroot);
617 extern int spa_vdev_attach(spa_t *spa, uint64_t guid, nvlist_t *nvroot,
618     int replacing);
619 extern int spa_vdev_detach(spa_t *spa, uint64_t guid, uint64_t pguid,
620     int replace_done);
621 extern int spa_vdev_remove(spa_t *spa, uint64_t guid, boolean_t unspare);
622 extern boolean_t spa_vdev_remove_active(spa_t *spa);
623 extern int spa_vdev_setpath(spa_t *spa, uint64_t guid, const char *newpath);
624 extern int spa_vdev_setfru(spa_t *spa, uint64_t guid, const char *newfru);
625 extern int spa_vdev_split_mirror(spa_t *spa, char *newname, nvlist_t *config,
626     nvlist_t *props, boolean_t exp);

628 /* spare state (which is global across all pools) */
629 extern void spa_spare_add(vdev_t *vd);
630 extern void spa_spare_remove(vdev_t *vd);
631 extern boolean_t spa_spare_exists(uint64_t guid, uint64_t *pool, int *refcnt);
632 extern void spa_spare_activate(vdev_t *vd);

634 /* L2ARC state (which is global across all pools) */
635 extern void spa_l2cache_add(vdev_t *vd);
636 extern void spa_l2cache_remove(vdev_t *vd);
637 extern boolean_t spa_l2cache_exists(uint64_t guid, uint64_t *pool);
638 extern void spa_l2cache_activate(vdev_t *vd);
```



```
639 extern void spa_l2cache_drop(spa_t *spa);

641 /* scanning */
642 extern int spa_scan(spa_t *spa, pool_scan_func_t func);
643 extern int spa_scan_stop(spa_t *spa);

645 /* spa syncing */
646 extern void spa_sync(spa_t *spa, uint64_t txg); /* only for DMU use */
647 extern void spa_sync_allpools(void);

649 /* spa namespace global mutex */
650 extern kmutex_t spa_namespace_lock;

652 /*
653  * SPA configuration functions in spa_config.c
654  */

656 #define SPA_CONFIG_UPDATE_POOL  0
657 #define SPA_CONFIG_UPDATE_VDEVS 1

659 extern void spa_config_sync(spa_t *, boolean_t, boolean_t);
660 extern void spa_config_load(void);
661 extern nvlist_t *spa_all_configs(uint64_t *);
662 extern void spa_config_set(spa_t *spa, nvlist_t *config);
663 extern nvlist_t *spa_config_generate(spa_t *spa, vdev_t *vd, uint64_t txg,
664     int getstats);
665 extern void spa_config_update(spa_t *spa, int what);

667 /*
668  * Miscellaneous SPA routines in spa_misc.c
669  */

671 /* Namespace manipulation */
672 extern spa_t *spa_lookup(const char *name);
673 extern spa_t *spa_add(const char *name, nvlist_t *config, const char *altroot);
674 extern void spa_remove(spa_t *spa);
675 extern spa_t *spa_next(spa_t *prev);

677 /* Refcount functions */
678 extern void spa_open_ref(spa_t *spa, void *tag);
679 extern void spa_close(spa_t *spa, void *tag);
680 extern boolean_t spa_refcount_zero(spa_t *spa);

682 #define SCL_NONE        0x00
683 #define SCL_CONFIG      0x01
684 #define SCL_STATE       0x02
685 #define SCL_L2ARC       0x04            /* hack until L2ARC 2.0 */
686 #define SCL_ALLOC       0x08
687 #define SCL_ZIO         0x10
688 #define SCL_FREE        0x20
689 #define SCL_VDEV        0x40
690 #define SCL_LOCKS       7
691 #define SCL_ALL         ((1 << SCL_LOCKS) - 1)
692 #define SCL_STATE_ALL   (SCL_STATE | SCL_L2ARC | SCL_ZIO)

694 /* Pool configuration locks */
695 extern int spa_config_tryenter(spa_t *spa, int locks, void *tag, krw_t rw);
696 extern void spa_config_enter(spa_t *spa, int locks, void *tag, krw_t rw);
697 extern void spa_config_exit(spa_t *spa, int locks, void *tag);
698 extern int spa_config_held(spa_t *spa, int locks, krw_t rw);

700 /* Pool vdev add/remove lock */
701 extern uint64_t spa_vdev_enter(spa_t *spa);
702 extern uint64_t spa_vdev_config_enter(spa_t *spa);
703 extern void spa_vdev_config_exit(spa_t *spa, vdev_t *vd, uint64_t txg,
704     int error, char *tag);
```

```
705 extern int spa_vdev_exit(spa_t *spa, vdev_t *vd, uint64_t txg, int error);

707 /* Pool vdev state change lock */
708 extern void spa_vdev_state_enter(spa_t *spa, int oplock);
709 extern int spa_vdev_state_exit(spa_t *spa, vdev_t *vd, int error);

711 /* Log state */
712 typedef enum spa_log_state {
713         SPA_LOG_UNKNOWN = 0,    /* unknown log state */
714         SPA_LOG_MISSING,        /* missing log(s) */
715         SPA_LOG_CLEAR,          /* clear the log(s) */
716         SPA_LOG_GOOD,           /* log(s) are good */
717 } spa_log_state_t;

719 extern spa_log_state_t spa_get_log_state(spa_t *spa);
720 extern void spa_set_log_state(spa_t *spa, spa_log_state_t state);
721 extern int spa_offline_log(spa_t *spa);

723 /* Log claim callback */
724 extern void spa_claim_notify(zio_t *zio);

726 /* Accessor functions */
727 extern boolean_t spa_shutting_down(spa_t *spa);
728 extern struct dsl_pool *spa_get_dsl(spa_t *spa);
729 extern boolean_t spa_is_initializing(spa_t *spa);
730 extern blkptr_t *spa_get_rootblkptr(spa_t *spa);
731 extern void spa_set_rootblkptr(spa_t *spa, const blkptr_t *bp);
732 extern void spa_altroot(spa_t *, char *, size_t);
733 extern int spa_sync_pass(spa_t *spa);
734 extern char *spa_name(spa_t *spa);
735 extern uint64_t spa_guid(spa_t *spa);
736 extern uint64_t spa_load_guid(spa_t *spa);
737 extern uint64_t spa_last_synced_txg(spa_t *spa);
738 extern uint64_t spa_first_txg(spa_t *spa);
739 extern uint64_t spa_syncing_txg(spa_t *spa);
740 extern uint64_t spa_version(spa_t *spa);
741 extern pool_state_t spa_state(spa_t *spa);
742 extern spa_load_state_t spa_load_state(spa_t *spa);
743 extern uint64_t spa_freeze_txg(spa_t *spa);
744 extern uint64_t spa_get_asize(spa_t *spa, uint64_t lsize);
745 extern uint64_t spa_get_dspace(spa_t *spa);
746 extern uint64_t spa_get_slop_space(spa_t *spa);
747 extern void spa_update_dspace(spa_t *spa);
748 extern uint64_t spa_version(spa_t *spa);
749 extern boolean_t spa_deflate(spa_t *spa);
750 extern metaslab_class_t *spa_normal_class(spa_t *spa);
751 extern metaslab_class_t *spa_log_class(spa_t *spa);
752 extern int spa_max_replication(spa_t *spa);
753 extern int spa_prev_software_version(spa_t *spa);
754 extern int spa_busy(void);
755 extern uint8_t spa_get_failmode(spa_t *spa);
756 extern boolean_t spa_suspended(spa_t *spa);
757 extern uint64_t spa_bootfs(spa_t *spa);
758 extern uint64_t spa_delegation(spa_t *spa);
759 extern objset_t *spa_meta_objset(spa_t *spa);
760 extern uint64_t spa_deadman_synctime(spa_t *spa);

762 /* Miscellaneous support routines */
763 extern void spa_activate_mos_feature(spa_t *spa, const char *feature,
764     dmu_tx_t *tx);
765 extern void spa_deactivate_mos_feature(spa_t *spa, const char *feature);
766 extern int spa_rename(const char *oldname, const char *newname);
767 extern spa_t *spa_by_guid(uint64_t pool_guid, uint64_t device_guid);
768 extern boolean_t spa_guid_exists(uint64_t pool_guid, uint64_t device_guid);
769 extern char *spa_strdup(const char *);
770 extern void spa_strfree(char *);
```



```
771 extern uint64_t spa_get_random(uint64_t range);
772 extern uint64_t spa_generate_guid(spa_t *spa);
773 extern void snprintf_blkptr(char *buf, size_t buflen, const blkptr_t *bp);
774 extern void spa_freeze(spa_t *spa);
775 extern int spa_change_guid(spa_t *spa);
776 extern void spa_upgrade(spa_t *spa, uint64_t version);
777 extern void spa_evict_all(void);
778 extern vdev_t *spa_lookup_by_guid(spa_t *spa, uint64_t guid,
779     boolean_t l2cache);
780 extern boolean_t spa_has_spare(spa_t *, uint64_t guid);
781 extern uint64_t dva_get_dsize_sync(spa_t *spa, const dva_t *dva);
782 extern uint64_t bp_get_dsize_sync(spa_t *spa, const blkptr_t *bp);
783 extern uint64_t bp_get_dsize(spa_t *spa, const blkptr_t *bp);
784 extern boolean_t spa_has_slogs(spa_t *spa);
785 extern boolean_t spa_is_root(spa_t *spa);
786 extern boolean_t spa_writeable(spa_t *spa);
787 extern boolean_t spa_has_pending_synctask(spa_t *spa);

789 extern int spa_mode(spa_t *spa);
790 extern uint64_t strtonum(const char *str, char **nptr);

792 extern char *spa_his_ievent_table[];

794 extern void spa_history_create_obj(spa_t *spa, dmu_tx_t *tx);
795 extern int spa_history_get(spa_t *spa, uint64_t *offset, uint64_t *len_read,
796     char *his_buf);
797 extern int spa_history_log(spa_t *spa, const char *his_buf);
798 extern int spa_history_log_nvl(spa_t *spa, nvlist_t *nvl);
799 extern void spa_history_log_version(spa_t *spa, const char *operation);
800 extern void spa_history_log_internal(spa_t *spa, const char *operation,
801     dmu_tx_t *tx, const char *fmt, ...);
802 extern void spa_history_log_internal_ds(struct dsl_dataset *ds, const char *op,
803     dmu_tx_t *tx, const char *fmt, ...);
804 extern void spa_history_log_internal_dd(dsl_dir_t *dd, const char *operation,
805     dmu_tx_t *tx, const char *fmt, ...);

807 /* error handling */
808 struct zbookmark_phys;
809 extern void spa_log_error(spa_t *spa, zio_t *zio);
810 extern void zfs_ereport_post(const char *class, spa_t *spa, vdev_t *vd,
811     zio_t *zio, uint64_t stateoroffset, uint64_t length);
812 extern void zfs_post_remove(spa_t *spa, vdev_t *vd);
813 extern void zfs_post_state_change(spa_t *spa, vdev_t *vd);
814 extern void zfs_post_autoreplace(spa_t *spa, vdev_t *vd);
815 extern uint64_t spa_get_errlog_size(spa_t *spa);
816 extern int spa_get_errlog(spa_t *spa, void *uaddr, size_t *count);
817 extern void spa_errlog_rotate(spa_t *spa);
818 extern void spa_errlog_drain(spa_t *spa);
819 extern void spa_errlog_sync(spa_t *spa, uint64_t txg);
820 extern void spa_get_errlists(spa_t *spa, avl_tree_t *last, avl_tree_t *scrub);

822 /* vdev cache */
823 extern void vdev_cache_stat_init(void);
824 extern void vdev_cache_stat_fini(void);

826 /* Initialization and termination */
827 extern void spa_init(int flags);
828 extern void spa_fini(void);
829 extern void spa_boot_init();

831 /* properties */
832 extern int spa_prop_set(spa_t *spa, nvlist_t *nvp);
833 extern int spa_prop_get(spa_t *spa, nvlist_t **nvp);
834 extern void spa_prop_clear_bootfs(spa_t *spa, uint64_t obj, dmu_tx_t *tx);
835 extern void spa_configfile_set(spa_t *, nvlist_t *, boolean_t);
```

```
837 /* asynchronous event notification */
838 extern void spa_event_notify(spa_t *spa, vdev_t *vdev, const char *name);

840 #ifdef ZFS_DEBUG
841 #define dprintf_bp(bp, fmt, ...) do {                           \
842         if (zfs_flags & ZFS_DEBUG_DPRINTF) {                    \
843         char *__blkbuf = kmem_alloc(BP_SPRINTF_LEN, KM_SLEEP);  \
844         snprintf_blkptr(__blkbuf, BP_SPRINTF_LEN, (bp));        \
845         dprintf(fmt " %s\n", __VA_ARGS__, __blkbuf);            \
846         kmem_free(__blkbuf, BP_SPRINTF_LEN);                    \
847         } \
848 _NOTE(CONSTCOND) } while (0)
849 #else
850 #define dprintf_bp(bp, fmt, ...)
851 #endif

853 extern boolean_t spa_debug_enabled(spa_t *spa);
854 #define spa_dbgmsg(spa, ...)                    \
855 {                                               \
856         if (spa_debug_enabled(spa))             \
857                 zfs_dbgmsg(__VA_ARGS__);        \
858 }

860 extern int spa_mode_global;                     /* mode, e.g. FREAD | FWRITE */

862 #ifdef  __cplusplus
863 }
864 #endif

866 #endif  /* _SYS_SPA_H */
```

```
**********************************************************
   18191 Tue Oct 28 11:57:19 2014
new/usr/src/uts/common/fs/zfs/sys/zio.h
Possibility to physically reserve space without writing leaf blocks
**********************************************************
______unchanged_portion_omitted_

 283 #define ZB_DESTROYED_OBJSET     (-1ULL)

 285 #define ZB_ROOT_OBJECT          (0ULL)
 286 #define ZB_ROOT_LEVEL           (-1LL)
 287 #define ZB_ROOT_BLKID           (0ULL)

 289 #define ZB_ZIL_OBJECT           (0ULL)
 290 #define ZB_ZIL_LEVEL            (-2LL)

 292 #define ZB_IS_ZERO(zb)                                          \
 293         ((zb)->zb_objset == 0 && (zb)->zb_object == 0 &&        \
 294         (zb)->zb_level == 0 && (zb)->zb_blkid == 0)
 295 #define ZB_IS_ROOT(zb)                          \
 296         ((zb)->zb_object == ZB_ROOT_OBJECT &&   \
 297         (zb)->zb_level == ZB_ROOT_LEVEL &&      \
 298         (zb)->zb_blkid == ZB_ROOT_BLKID)

 300 typedef struct zio_prop {
 301         enum zio_checksum       zp_checksum;
 302         enum zio_compress       zp_compress;
 303         dmu_object_type_t       zp_type;
 304         uint8_t                 zp_level;
 305         uint8_t                 zp_copies;
 306         boolean_t               zp_dedup;
 307         boolean_t               zp_dedup_verify;
 308         boolean_t               zp_nopwrite;
 309         boolean_t               zp_zero_write;
 310 #endif /* ! codereview */
 311 } zio_prop_t;

 313 typedef struct zio_cksum_report zio_cksum_report_t;

 315 typedef void zio_cksum_finish_f(zio_cksum_report_t *rep,
 316     const void *good_data);
 317 typedef void zio_cksum_free_f(void *cbdata, size_t size);

 319 struct zio_bad_cksum;                           /* defined in zio_checksum.h */
 320 struct dnode_phys;

 322 struct zio_cksum_report {
 323         struct zio_cksum_report *zcr_next;
 324         nvlist_t                *zcr_ereport;
 325         nvlist_t                *zcr_detector;
 326         void                    *zcr_cbdata;
 327         size_t                  zcr_cbinfo;     /* passed to zcr_free() */
 328         uint64_t                zcr_align;
 329         uint64_t                zcr_length;
 330         zio_cksum_finish_f      *zcr_finish;
 331         zio_cksum_free_f        *zcr_free;

 333         /* internal use only */
 334         struct zio_bad_cksum    *zcr_ckinfo;    /* information from failure */
 335 };

 337 typedef void zio_vsd_cksum_report_f(zio_t *zio, zio_cksum_report_t *zcr,
 338     void *arg);

 340 zio_vsd_cksum_report_f  zio_vsd_default_cksum_report;
```



```
 342 typedef struct zio_vsd_ops {
 343         zio_done_func_t         *vsd_free;
 344         zio_vsd_cksum_report_f  *vsd_cksum_report;
 345 } zio_vsd_ops_t;

 347 typedef struct zio_gang_node {
 348         zio_gbh_phys_t          *gn_gbh;
 349         struct zio_gang_node    *gn_child[SPA_GBH_NBLKPTRS];
 350 } zio_gang_node_t;

 352 typedef zio_t *zio_gang_issue_func_t(zio_t *zio, blkptr_t *bp,
 353     zio_gang_node_t *gn, void *data);

 355 typedef void zio_transform_func_t(zio_t *zio, void *data, uint64_t size);

 357 typedef struct zio_transform {
 358         void                    *zt_orig_data;
 359         uint64_t                zt_orig_size;
 360         uint64_t                zt_bufsize;
 361         zio_transform_func_t    *zt_transform;
 362         struct zio_transform    *zt_next;
 363 } zio_transform_t;

 365 typedef int zio_pipe_stage_t(zio_t *zio);

 367 /*
 368  * The io_reexecute flags are distinct from io_flags because the child must
 369  * be able to propagate them to the parent.  The normal io_flags are local
 370  * to the zio, not protected by any lock, and not modifiable by children;
 371  * the reexecute flags are protected by io_lock, modifiable by children,
 372  * and always propagated -- even when ZIO_FLAG_DONT_PROPAGATE is set.
 373  */
 374 #define ZIO_REEXECUTE_NOW       0x01
 375 #define ZIO_REEXECUTE_SUSPEND   0x02

 377 typedef struct zio_link {
 378         zio_t           *zl_parent;
 379         zio_t           *zl_child;
 380         list_node_t     zl_parent_node;
 381         list_node_t     zl_child_node;
 382 } zio_link_t;

 384 struct zio {
 385         /* Core information about this I/O */
 386         zbookmark_phys_t        io_bookmark;
 387         zio_prop_t      io_prop;
 388         zio_type_t      io_type;
 389         enum zio_child  io_child_type;
 390         int             io_cmd;
 391         zio_priority_t  io_priority;
 392         uint8_t         io_reexecute;
 393         uint8_t         io_state[ZIO_WAIT_TYPES];
 394         uint64_t        io_txg;
 395         spa_t           *io_spa;
 396         blkptr_t        *io_bp;
 397         blkptr_t        *io_bp_override;
 398         blkptr_t        io_bp_copy;
 399         list_t          io_parent_list;
 400         list_t          io_child_list;
 401         zio_link_t      *io_walk_link;
 402         zio_t           *io_logical;
 403         zio_transform_t *io_transform_stack;

 405         /* Callback info */
 406         zio_done_func_t *io_ready;
 407         zio_done_func_t *io_physdone;
```

```
408         zio_done_func_t *io_done;
409         void            *io_private;
410         int64_t         io_prev_space_delta;    /* DMU private */
411         blkptr_t        io_bp_orig;

413         /* Data represented by this I/O */
414         void            *io_data;
415         void            *io_orig_data;
416         uint64_t        io_size;
417         uint64_t        io_orig_size;

419         /* Stuff for the vdev stack */
420         vdev_t          *io_vd;
421         void            *io_vsd;
422         const zio_vsd_ops_t *io_vsd_ops;

424         uint64_t        io_offset;
425         hrtime_t        io_timestamp;
426         avl_node_t      io_queue_node;

428         /* Internal pipeline state */
429         enum zio_flag   io_flags;
430         enum zio_stage  io_stage;
431         enum zio_stage  io_pipeline;
432         enum zio_flag   io_orig_flags;
433         enum zio_stage  io_orig_stage;
434         enum zio_stage  io_orig_pipeline;
435         int             io_error;
436         int             io_child_error[ZIO_CHILD_TYPES];
437         uint64_t        io_children[ZIO_CHILD_TYPES][ZIO_WAIT_TYPES];
438         uint64_t        io_child_count;
439         uint64_t        io_phys_children;
440         uint64_t        io_parent_count;
441         uint64_t        *io_stall;
442         zio_t           *io_gang_leader;
443         zio_gang_node_t *io_gang_tree;
444         void            *io_executor;
445         void            *io_waiter;
446         kmutex_t        io_lock;
447         kcondvar_t      io_cv;

449         /* FMA state */
450         zio_cksum_report_t *io_cksum_report;
451         uint64_t        io_ena;

453         /* Taskq dispatching state */
454         taskq_ent_t     io_tqent;
455 };

457 extern zio_t *zio_null(zio_t *pio, spa_t *spa, vdev_t *vd,
458     zio_done_func_t *done, void *private, enum zio_flag flags);

460 extern zio_t *zio_root(spa_t *spa,
461     zio_done_func_t *done, void *private, enum zio_flag flags);

463 extern zio_t *zio_read(zio_t *pio, spa_t *spa, const blkptr_t *bp, void *data,
464     uint64_t size, zio_done_func_t *done, void *private,
465     zio_priority_t priority, enum zio_flag flags, const zbookmark_phys_t *zb);

467 extern zio_t *zio_write(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp,
468     void *data, uint64_t size, const zio_prop_t *zp,
469     zio_done_func_t *ready, zio_done_func_t *physdone, zio_done_func_t *done,
470     void *private,
471     zio_priority_t priority, enum zio_flag flags, const zbookmark_phys_t *zb);

473 extern zio_t *zio_rewrite(zio_t *pio, spa_t *spa, uint64_t txg, blkptr_t *bp,
```



```
474     void *data, uint64_t size, zio_done_func_t *done, void *private,
475     zio_priority_t priority, enum zio_flag flags, zbookmark_phys_t *zb);

477 extern void zio_write_override(zio_t *zio, blkptr_t *bp, int copies,
478     boolean_t nopwrite);

480 extern void zio_free(spa_t *spa, uint64_t txg, const blkptr_t *bp);

482 extern zio_t *zio_claim(zio_t *pio, spa_t *spa, uint64_t txg,
483     const blkptr_t *bp,
484     zio_done_func_t *done, void *private, enum zio_flag flags);

486 extern zio_t *zio_ioctl(zio_t *pio, spa_t *spa, vdev_t *vd, int cmd,
487     zio_done_func_t *done, void *private, enum zio_flag flags);

489 extern zio_t *zio_read_phys(zio_t *pio, vdev_t *vd, uint64_t offset,
490     uint64_t size, void *data, int checksum,
491     zio_done_func_t *done, void *private, zio_priority_t priority,
492     enum zio_flag flags, boolean_t labels);

494 extern zio_t *zio_write_phys(zio_t *pio, vdev_t *vd, uint64_t offset,
495     uint64_t size, void *data, int checksum,
496     zio_done_func_t *done, void *private, zio_priority_t priority,
497     enum zio_flag flags, boolean_t labels);

499 extern zio_t *zio_free_sync(zio_t *pio, spa_t *spa, uint64_t txg,
500     const blkptr_t *bp, enum zio_flag flags);

502 extern int zio_alloc_zil(spa_t *spa, uint64_t txg, blkptr_t *new_bp,
503     blkptr_t *old_bp, uint64_t size, boolean_t use_slog);
504 extern void zio_free_zil(spa_t *spa, uint64_t txg, blkptr_t *bp);
505 extern void zio_flush(zio_t *zio, vdev_t *vd);
506 extern void zio_shrink(zio_t *zio, uint64_t size);

508 extern int zio_wait(zio_t *zio);
509 extern void zio_nowait(zio_t *zio);
510 extern void zio_execute(zio_t *zio);
511 extern void zio_interrupt(zio_t *zio);

513 extern zio_t *zio_walk_parents(zio_t *cio);
514 extern zio_t *zio_walk_children(zio_t *pio);
515 extern zio_t *zio_unique_parent(zio_t *cio);
516 extern void zio_add_child(zio_t *pio, zio_t *cio);

518 extern void *zio_buf_alloc(size_t size);
519 extern void zio_buf_free(void *buf, size_t size);
520 extern void *zio_data_buf_alloc(size_t size);
521 extern void zio_data_buf_free(void *buf, size_t size);

523 extern void zio_resubmit_stage_async(void *);

525 extern zio_t *zio_vdev_child_io(zio_t *zio, blkptr_t *bp, vdev_t *vd,
526     uint64_t offset, void *data, uint64_t size, int type,
527     zio_priority_t priority, enum zio_flag flags,
528     zio_done_func_t *done, void *private);

530 extern zio_t *zio_vdev_delegated_io(vdev_t *vd, uint64_t offset,
531     void *data, uint64_t size, int type, zio_priority_t priority,
532     enum zio_flag flags, zio_done_func_t *done, void *private);

534 extern void zio_vdev_io_bypass(zio_t *zio);
535 extern void zio_vdev_io_reissue(zio_t *zio);
536 extern void zio_vdev_io_redone(zio_t *zio);

538 extern void zio_checksum_verified(zio_t *zio);
539 extern int zio_worst_error(int e1, int e2);
```

```
extern enum zio_checksum zio_checksum_select(enum zio_checksum child,
    enum zio_checksum parent);
extern enum zio_checksum zio_checksum_dedup_select(spa_t *spa,
    enum zio_checksum child, enum zio_checksum parent);
extern enum zio_compress zio_compress_select(enum zio_compress child,
    enum zio_compress parent);

extern void zio_suspend(spa_t *spa, zio_t *zio);
extern int zio_resume(spa_t *spa);
extern void zio_resume_wait(spa_t *spa);

/*
 * Initial setup and teardown.
 */
extern void zio_init(void);
extern void zio_fini(void);

/*
 * Fault injection
 */
struct zinject_record;
extern uint32_t zio_injection_enabled;
extern int zio_inject_fault(char *name, int flags, int *id,
    struct zinject_record *record);
extern int zio_inject_list_next(int *id, char *name, size_t buflen,
    struct zinject_record *record);
extern int zio_clear_fault(int id);
extern void zio_handle_panic_injection(spa_t *spa, char *tag, uint64_t type);
extern int zio_handle_fault_injection(zio_t *zio, int error);
extern int zio_handle_device_injection(vdev_t *vd, zio_t *zio, int error);
extern int zio_handle_label_injection(zio_t *zio, int error);
extern void zio_handle_ignored_writes(zio_t *zio);
extern uint64_t zio_handle_io_delay(zio_t *zio);

/*
 * Checksum ereport functions
 */
extern void zfs_ereport_start_checksum(spa_t *spa, vdev_t *vd, struct zio *zio,
    uint64_t offset, uint64_t length, void *arg, struct zio_bad_cksum *info);
extern void zfs_ereport_finish_checksum(zio_cksum_report_t *report,
    const void *good_data, const void *bad_data, boolean_t drop_if_identical);

extern void zfs_ereport_send_interim_checksum(zio_cksum_report_t *report);
extern void zfs_ereport_free_checksum(zio_cksum_report_t *report);

/* If we have the good data in hand, this function can be used */
extern void zfs_ereport_post_checksum(spa_t *spa, vdev_t *vd,
    struct zio *zio, uint64_t offset, uint64_t length,
    const void *good_data, const void *bad_data, struct zio_bad_cksum *info);

/* Called from spa_sync(), but primarily an injection handler */
extern void spa_handle_ignored_writes(spa_t *spa);

/* zbookmark_phys functions */
boolean_t zbookmark_is_before(const struct dnode_phys *dnp,
    const zbookmark_phys_t *zb1, const zbookmark_phys_t *zb2);

#ifdef  __cplusplus
}
#endif

#endif  /* _ZIO_H */
```

```
**********************************************************
  135331 Tue Oct 28 11:57:20 2014
new/usr/src/uts/common/fs/zfs/zfs_vnops.c
Possibility to physically reserve space without writing leaf blocks
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2012, 2014 by Delphix. All rights reserved.
 * Copyright 2014 Nexenta Systems, Inc.  All rights reserved.
 */

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

#include <sys/types.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/resource.h>
#include <sys/vfs.h>
#include <sys/vfs_opreg.h>
#include <sys/vnode.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/kmem.h>
#include <sys/taskq.h>
#include <sys/uio.h>
#include <sys/vmsystm.h>
#include <sys/atomic.h>
#include <sys/vm.h>
#include <vm/seg_vn.h>
#include <vm/pvn.h>
#include <vm/as.h>
#include <vm/kpm.h>
#include <vm/seg_kpm.h>
#include <sys/mman.h>
#include <sys/pathname.h>
#include <sys/cmn_err.h>
#include <sys/errno.h>
#include <sys/unistd.h>
#include <sys/zfs_dir.h>
#include <sys/zfs_acl.h>
#include <sys/zfs_ioctl.h>
#include <sys/fs/zfs.h>
#include <sys/dmu.h>
```



```
#include <sys/dmu_objset.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/dbuf.h>
#include <sys/zap.h>
#include <sys/sa.h>
#include <sys/dirent.h>
#include <sys/policy.h>
#include <sys/sunddi.h>
#include <sys/filio.h>
#include <sys/sid.h>
#include "fs/fs_subr.h"
#include <sys/zfs_ctldir.h>
#include <sys/zfs_fuid.h>
#include <sys/zfs_sa.h>
#include <sys/zfeature.h>
#endif /* ! codereview */
#include <sys/dnlc.h>
#include <sys/zfs_rlock.h>
#include <sys/extdirent.h>
#include <sys/kidmap.h>
#include <sys/cred.h>
#include <sys/attr.h>

/*
 * Programming rules.
 *
 * Each vnode op performs some logical unit of work.  To do this, the ZPL must
 * properly lock its in-core state, create a DMU transaction, do the work,
 * record this work in the intent log (ZIL), commit the DMU transaction,
 * and wait for the intent log to commit if it is a synchronous operation.
 * Moreover, the vnode ops must work in both normal and log replay context.
 * The ordering of events is important to avoid deadlocks and references
 * to freed memory.  The example below illustrates the following Big Rules:
 *
 *  (1) A check must be made in each zfs thread for a mounted file system.
 *      This is done avoiding races using ZFS_ENTER(zfsvfs).
 *      A ZFS_EXIT(zfsvfs) is needed before all returns.  Any znodes
 *      must be checked with ZFS_VERIFY_ZP(zp).  Both of these macros
 *      can return EIO from the calling function.
 *
 *  (2) VN_RELE() should always be the last thing except for zil_commit()
 *      (if necessary) and ZFS_EXIT(). This is for 3 reasons:
 *      First, if it's the last reference, the vnode/znode
 *      can be freed, so the zp may point to freed memory.  Second, the last
 *      reference will call zfs_zinactive(), which may induce a lot of work --
 *      pushing cached pages (which acquires range locks) and syncing out
 *      cached atime changes.  Third, zfs_zinactive() may require a new tx,
 *      which could deadlock the system if you were already holding one.
 *      If you must call VN_RELE() within a tx then use VN_RELE_ASYNC().
 *
 *  (3) All range locks must be grabbed before calling dmu_tx_assign(),
 *      as they can span dmu_tx_assign() calls.
 *
 *  (4) If ZPL locks are held, pass TXG_NOWAIT as the second argument to
 *      dmu_tx_assign().  This is critical because we don't want to block
 *      while holding locks.
 *
 *      If no ZPL locks are held (aside from ZFS_ENTER()), use TXG_WAIT.  This
 *      reduces lock contention and CPU usage when we must wait (note that if
 *      throughput is constrained by the storage, nearly every transaction
 *      must wait).
 *
 *      Note, in particular, that if a lock is sometimes acquired before
 *      the tx assigns, and sometimes after (e.g. z_lock), then failing
 *      to use a non-blocking assign can deadlock the system.  The scenario:
```

```
128  *
129  *      Thread A has grabbed a lock before calling dmu_tx_assign().
130  *      Thread B is in an already-assigned tx, and blocks for this lock.
131  *      Thread A calls dmu_tx_assign(TXG_WAIT) and blocks in txg_wait_open()
132  *      forever, because the previous txg can't quiesce until B's tx commits.
133  *
134  *      If dmu_tx_assign() returns ERESTART and zfsvfs->z_assign is TXG_NOWAIT,
135  *      then drop all locks, call dmu_tx_wait(), and try again.  On subsequent
136  *      calls to dmu_tx_assign(), pass TXG_WAITED rather than TXG_NOWAIT,
137  *      to indicate that this operation has already called dmu_tx_wait().
138  *      This will ensure that we don't retry forever, waiting a short bit
139  *      each time.
140  *
141  *  (5) If the operation succeeded, generate the intent log entry for it
142  *      before dropping locks.  This ensures that the ordering of events
143  *      in the intent log matches the order in which they actually occurred.
144  *      During ZIL replay the zfs_log_* functions will update the sequence
145  *      number to indicate the zil transaction has replayed.
146  *
147  *  (6) At the end of each vnode op, the DMU tx must always commit,
148  *      regardless of whether there were any errors.
149  *
150  *  (7) After dropping all locks, invoke zil_commit(zilog, foid)
151  *      to ensure that synchronous semantics are provided when necessary.
152  *
153  * In general, this is how things should be ordered in each vnode op:
154  *
155  *      ZFS_ENTER(zfsvfs);              // exit if unmounted
156  * top:
157  *      zfs_dirent_lock(&dl, ...)       // lock directory entry (may VN_HOLD())
158  *      rw_enter(...);                  // grab any other locks you need
159  *      tx = dmu_tx_create(...);        // get DMU tx
160  *      dmu_tx_hold_*();                // hold each object you might modify
161  *      error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
162  *      if (error) {
163  *              rw_exit(...);           // drop locks
164  *              zfs_dirent_unlock(dl);  // unlock directory entry
165  *              VN_RELE(...);           // release held vnodes
166  *              if (error == ERESTART) {
167  *                      waited = B_TRUE;
168  *                      dmu_tx_wait(tx);
169  *                      dmu_tx_abort(tx);
170  *                      goto top;
171  *              }
172  *              dmu_tx_abort(tx);       // abort DMU tx
173  *              ZFS_EXIT(zfsvfs);       // finished in zfs
174  *              return (error);         // really out of space
175  *      }
176  *      error = do_real_work();         // do whatever this VOP does
177  *      if (error == 0)
178  *              zfs_log_*(...);         // on success, make ZIL entry
179  *      dmu_tx_commit(tx);              // commit DMU tx -- error or not
180  *      rw_exit(...);                   // drop locks
181  *      zfs_dirent_unlock(dl);          // unlock directory entry
182  *      VN_RELE(...);                   // release held vnodes
183  *      zil_commit(zilog, foid);        // synchronous when necessary
184  *      ZFS_EXIT(zfsvfs);               // finished in zfs
185  *      return (error);                 // done, report error
186  */

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



```
195         ZFS_ENTER(zfsvfs);
196         ZFS_VERIFY_ZP(zp);

198         if ((flag & FWRITE) && (zp->z_pflags & ZFS_APPENDONLY) &&
199             ((flag & FAPPEND) == 0)) {
200                 ZFS_EXIT(zfsvfs);
201                 return (SET_ERROR(EPERM));
202         }

204         if (!zfs_has_ctldir(zp) && zp->z_zfsvfs->z_vscan &&
205             ZTOV(zp)->v_type == VREG &&
206             !(zp->z_pflags & ZFS_AV_QUARANTINED) && zp->z_size > 0) {
207                 if (fs_vscan(*vpp, cr, 0) != 0) {
208                         ZFS_EXIT(zfsvfs);
209                         return (SET_ERROR(EACCES));
210                 }
211         }

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

217         ZFS_EXIT(zfsvfs);
218         return (0);
219 }

221 /* ARGSUSED */
222 static int
223 zfs_close(vnode_t *vp, int flag, int count, offset_t offset, cred_t *cr,
224     caller_context_t *ct)
225 {
226         znode_t *zp = VTOZ(vp);
227         zfsvfs_t *zfsvfs = zp->z_zfsvfs;

229         /*
230          * Clean up any locks held by this process on the vp.
231          */
232         cleanlocks(vp, ddi_get_pid(), 0);
233         cleanshares(vp, ddi_get_pid());

235         ZFS_ENTER(zfsvfs);
236         ZFS_VERIFY_ZP(zp);

238         /* Decrement the synchronous opens in the znode */
239         if ((flag & (FSYNC | FDSYNC)) && (count == 1))
240                 atomic_dec_32(&zp->z_sync_cnt);

242         if (!zfs_has_ctldir(zp) && zp->z_zfsvfs->z_vscan &&
243             ZTOV(zp)->v_type == VREG &&
244             !(zp->z_pflags & ZFS_AV_QUARANTINED) && zp->z_size > 0)
245                 VERIFY(fs_vscan(vp, cr, 1) == 0);

247         ZFS_EXIT(zfsvfs);
248         return (0);
249 }

251 /*
252  * Lseek support for finding holes (cmd == _FIO_SEEK_HOLE) and
253  * data (cmd == _FIO_SEEK_DATA). "off" is an in/out parameter.
254  */
255 static int
256 zfs_holey(vnode_t *vp, int cmd, offset_t *off)
257 {
258         znode_t *zp = VTOZ(vp);
259         uint64_t noff = (uint64_t)*off; /* new offset */
```

```
260         uint64_t file_sz;
261         int error;
262         boolean_t hole;

264         file_sz = zp->z_size;
265         if (noff >= file_sz)  {
266                 return (SET_ERROR(ENXIO));
267         }

269         if (cmd == _FIO_SEEK_HOLE)
270                 hole = B_TRUE;
271         else
272                 hole = B_FALSE;

274         error = dmu_offset_next(zp->z_zfsvfs->z_os, zp->z_id, hole, &noff);

276         if (error == ESRCH)
277                 return (SET_ERROR(ENXIO));

279         /*
280          * We could find a hole that begins after the logical end-of-file,
281          * because dmu_offset_next() only works on whole blocks.  If the
282          * EOF falls mid-block, then indicate that the "virtual hole"
283          * at the end of the file begins at the logical EOF, rather than
284          * at the end of the last block.
285          */
286         if (noff > file_sz) {
287                 ASSERT(hole);
288                 noff = file_sz;
289         }

291         if (noff < *off)
292                 return (error);
293         *off = noff;
294         return (error);
295 }


298 static int zfs_zero_write(vnode_t *vp, uint64_t size, cred_t *cr,
299     caller_context_t *ct);

301 #endif /* ! codereview */
302 /* ARGSUSED */
303 static int
304 zfs_ioctl(vnode_t *vp, int com, intptr_t data, int flag, cred_t *cred,
305     int *rvalp, caller_context_t *ct)
306 {
307         offset_t off;
308         int error;
309         zfsvfs_t *zfsvfs;
310         znode_t *zp;
311         uint64_t size;
312 #endif /* ! codereview */

314         switch (com) {
315         case _FIOFFS:
316                 return (zfs_sync(vp->v_vfsp, 0, cred));

318                 /*
319                  * The following two ioctls are used by bfu.  Faking out,
320                  * necessary to avoid bfu errors.
321                  */
322         case _FIOGDIO:
323         case _FIOSDIO:
324                 return (0);
```



```
326         case _FIO_SEEK_DATA:
327         case _FIO_SEEK_HOLE:
328                 if (ddi_copyin((void *)data, &off, sizeof (off), flag))
329                         return (SET_ERROR(EFAULT));

331                 zp = VTOZ(vp);
332                 zfsvfs = zp->z_zfsvfs;
333                 ZFS_ENTER(zfsvfs);
334                 ZFS_VERIFY_ZP(zp);

336                 /* offset parameter is in/out */
337                 error = zfs_holey(vp, com, &off);
338                 ZFS_EXIT(zfsvfs);
339                 if (error)
340                         return (error);
341                 if (ddi_copyout(&off, (void *)data, sizeof (off), flag))
342                         return (SET_ERROR(EFAULT));
343                 return (0);
344         case _FIO_RESERVE_SPACE:
345                 if (ddi_copyin((void *)data, &size, sizeof (size), flag))
346                         return (EFAULT);
347                 error = zfs_zero_write(vp, size, cred, ct);
348                 return (error);
349 #endif /* ! codereview */
350         }
351         return (SET_ERROR(ENOTTY));
352 }

354 /*
355  * Utility functions to map and unmap a single physical page.  These
356  * are used to manage the mappable copies of ZFS file data, and therefore
357  * do not update ref/mod bits.
358  */
359 caddr_t
360 zfs_map_page(page_t *pp, enum seg_rw rw)
361 {
362         if (kpm_enable)
363                 return (hat_kpm_mapin(pp, 0));
364         ASSERT(rw == S_READ || rw == S_WRITE);
365         return (ppmapin(pp, PROT_READ | ((rw == S_WRITE) ? PROT_WRITE : 0),
366             (caddr_t)-1));
367 }

369 void
370 zfs_unmap_page(page_t *pp, caddr_t addr)
371 {
372         if (kpm_enable) {
373                 hat_kpm_mapout(pp, 0, addr);
374         } else {
375                 ppmapout(addr);
376         }
377 }

379 /*
380  * When a file is memory mapped, we must keep the IO data synchronized
381  * between the DMU cache and the memory mapped pages.  What this means:
382  *
383  * On Write:    If we find a memory mapped page, we write to *both*
384  *              the page and the dmu buffer.
385  */
386 static void
387 update_pages(vnode_t *vp, int64_t start, int len, objset_t *os, uint64_t oid)
388 {
389         int64_t off;

391         off = start & PAGEOFFSET;
```

```
392         for (start &= PAGEMASK; len > 0; start += PAGESIZE) {
393                 page_t *pp;
394                 uint64_t nbytes = MIN(PAGESIZE - off, len);
396                 if (pp = page_lookup(vp, start, SE_SHARED)) {
397                         caddr_t va;
399                         va = zfs_map_page(pp, S_WRITE);
400                         (void) dmu_read(os, oid, start+off, nbytes, va+off,
401                             DMU_READ_PREFETCH);
402                         zfs_unmap_page(pp, va);
403                         page_unlock(pp);
404                 }
405                 len -= nbytes;
406                 off = 0;
407         }
408 }
410 /*
411  * When a file is memory mapped, we must keep the IO data synchronized
412  * between the DMU cache and the memory mapped pages.  What this means:
413  *
414  * On Read:     We "read" preferentially from memory mapped pages,
415  *              else we default from the dmu buffer.
416  *
417  * NOTE: We will always "break up" the IO into PAGESIZE uiomoves when
418  *       the file is memory mapped.
419  */
420 static int
421 mappedread(vnode_t *vp, int nbytes, uio_t *uio)
422 {
423         znode_t *zp = VTOZ(vp);
424         int64_t start, off;
425         int len = nbytes;
426         int error = 0;
428         start = uio->uio_loffset;
429         off = start & PAGEOFFSET;
430         for (start &= PAGEMASK; len > 0; start += PAGESIZE) {
431                 page_t *pp;
432                 uint64_t bytes = MIN(PAGESIZE - off, len);
434                 if (pp = page_lookup(vp, start, SE_SHARED)) {
435                         caddr_t va;
437                         va = zfs_map_page(pp, S_READ);
438                         error = uiomove(va + off, bytes, UIO_READ, uio);
439                         zfs_unmap_page(pp, va);
440                         page_unlock(pp);
441                 } else {
442                         error = dmu_read_uio_dbuf(sa_get_db(zp->z_sa_hdl),
443                             uio, bytes);
444                 }
445                 len -= bytes;
446                 off = 0;
447                 if (error)
448                         break;
449         }
450         return (error);
451 }
453 offset_t zfs_read_chunk_size = 1024 * 1024; /* Tunable */
455 /*
456  * Read bytes from specified file into supplied buffer.
457  *
```



```
458  *      IN:     vp      - vnode of file to be read from.
459  *              uio     - structure supplying read location, range info,
460  *                        and return buffer.
461  *              ioflag  - SYNC flags; used to provide FRSYNC semantics.
462  *              cr      - credentials of caller.
463  *              ct      - caller context
464  *
465  *      OUT:    uio     - updated offset and range, buffer filled.
466  *
467  *      RETURN: 0 on success, error code on failure.
468  *
469  * Side Effects:
470  *      vp - atime updated if byte count > 0
471  */
472 /* ARGSUSED */
473 static int
474 zfs_read(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cr, caller_context_t *ct)
475 {
476         znode_t         *zp = VTOZ(vp);
477         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
478         ssize_t         n, nbytes;
479         int             error = 0;
480         rl_t            *rl;
481         xuio_t          *xuio = NULL;
483         ZFS_ENTER(zfsvfs);
484         ZFS_VERIFY_ZP(zp);
486         if (zp->z_pflags & ZFS_AV_QUARANTINED) {
487                 ZFS_EXIT(zfsvfs);
488                 return (SET_ERROR(EACCES));
489         }
491         /*
492          * Validate file offset
493          */
494         if (uio->uio_loffset < (offset_t)0) {
495                 ZFS_EXIT(zfsvfs);
496                 return (SET_ERROR(EINVAL));
497         }
499         /*
500          * Fasttrack empty reads
501          */
502         if (uio->uio_resid == 0) {
503                 ZFS_EXIT(zfsvfs);
504                 return (0);
505         }
507         /*
508          * Check for mandatory locks
509          */
510         if (MANDMODE(zp->z_mode)) {
511                 if (error = chklock(vp, FREAD,
512                     uio->uio_loffset, uio->uio_resid, uio->uio_fmode, ct)) {
513                         ZFS_EXIT(zfsvfs);
514                         return (error);
515                 }
516         }
518         /*
519          * If we're in FRSYNC mode, sync out this znode before reading it.
520          */
521         if (ioflag & FRSYNC || zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
522                 zil_commit(zfsvfs->z_log, zp->z_id);
```

```
524         /*
525          * Lock the range against changes.
526          */
527         rl = zfs_range_lock(zp, uio->uio_loffset, uio->uio_resid, RL_READER);
528
529         /*
530          * If we are reading past end-of-file we can skip
531          * to the end; but we might still need to set atime.
532          */
533         if (uio->uio_loffset >= zp->z_size) {
534                 error = 0;
535                 goto out;
536         }
537
538         ASSERT(uio->uio_loffset < zp->z_size);
539         n = MIN(uio->uio_resid, zp->z_size - uio->uio_loffset);
540
541         if ((uio->uio_extflg == UIO_XUIO) &&
542             (((xuio_t *)uio)->xu_type == UIOTYPE_ZEROCOPY)) {
543                 int nblk;
544                 int blksz = zp->z_blksz;
545                 uint64_t offset = uio->uio_loffset;
546
547                 xuio = (xuio_t *)uio;
548                 if ((ISP2(blksz))) {
549                         nblk = (P2ROUNDUP(offset + n, blksz) - P2ALIGN(offset,
550                             blksz)) / blksz;
551                 } else {
552                         ASSERT(offset + n <= blksz);
553                         nblk = 1;
554                 }
555                 (void) dmu_xuio_init(xuio, nblk);
556
557                 if (vn_has_cached_data(vp)) {
558                         /*
559                          * For simplicity, we always allocate a full buffer
560                          * even if we only expect to read a portion of a block.
561                          */
562                         while (--nblk >= 0) {
563                                 (void) dmu_xuio_add(xuio,
564                                     dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
565                                     blksz), 0, blksz);
566                         }
567                 }
568         }
569
570         while (n > 0) {
571                 nbytes = MIN(n, zfs_read_chunk_size -
572                     P2PHASE(uio->uio_loffset, zfs_read_chunk_size));
573
574                 if (vn_has_cached_data(vp)) {
575                         error = mappedread(vp, nbytes, uio);
576                 } else {
577                         error = dmu_read_uio_dbuf(sa_get_db(zp->z_sa_hdl),
578                             uio, nbytes);
579                 }
580                 if (error) {
581                         /* convert checksum errors into IO errors */
582                         if (error == ECKSUM)
583                                 error = SET_ERROR(EIO);
584                         break;
585                 }
586
587                 n -= nbytes;
588         }
589 out:
```



```
590         zfs_range_unlock(rl);
591
592         ZFS_ACCESSTIME_STAMP(zfsvfs, zp);
593         ZFS_EXIT(zfsvfs);
594         return (error);
595 }
596
597 /*
598  * Write the bytes to a file.
599  *
600  *      IN:     vp      - vnode of file to be written to.
601  *              uio     - structure supplying write location, range info,
602  *                        and data buffer.
603  *              ioflag  - FAPPEND, FSYNC, and/or FDSYNC.  FAPPEND is
604  *                        set if in append mode.
605  *              cr      - credentials of caller.
606  *              ct      - caller context (NFS/CIFS fem monitor only)
607  *
608  *      OUT:    uio     - updated offset and range.
609  *
610  *      RETURN: 0 on success, error code on failure.
611  *
612  * Timestamps:
613  *      vp - ctime|mtime updated if byte count > 0
614  */
615
616 /* ARGSUSED */
617 static int
618 zfs_write(vnode_t *vp, uio_t *uio, int ioflag, cred_t *cr, caller_context_t *ct)
619 {
620         znode_t         *zp = VTOZ(vp);
621         rlim64_t        limit = uio->uio_llimit;
622         ssize_t         start_resid = uio->uio_resid;
623         ssize_t         tx_bytes;
624         uint64_t        end_size;
625         dmu_tx_t        *tx;
626         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
627         zilog_t         *zilog;
628         offset_t        woff;
629         ssize_t         n, nbytes;
630         rl_t            *rl;
631         int             max_blksz = zfsvfs->z_max_blksz;
632         int             error = 0;
633         arc_buf_t       *abuf;
634         iovec_t         *aiov = NULL;
635         xuio_t          *xuio = NULL;
636         int             i_iov = 0;
637         int             iovcnt = uio->uio_iovcnt;
638         iovec_t         *iovp = uio->uio_iov;
639         int             write_eof;
640         int             count = 0;
641         sa_bulk_attr_t  bulk[4];
642         uint64_t        mtime[2], ctime[2];
643
644         /*
645          * Fasttrack empty write
646          */
647         n = start_resid;
648         if (n == 0)
649                 return (0);
650
651         if (limit == RLIM64_INFINITY || limit > MAXOFFSET_T)
652                 limit = MAXOFFSET_T;
653
654         ZFS_ENTER(zfsvfs);
655         ZFS_VERIFY_ZP(zp);
```

```
657         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
658         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);
659         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_SIZE(zfsvfs), NULL,
660             &zp->z_size, 8);
661         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
662             &zp->z_pflags, 8);

664         /*
665          * In a case vp->v_vfsp != zp->z_zfsvfs->z_vfs (e.g. snapshots) our
666          * callers might not be able to detect properly that we are read-only,
667          * so check it explicitly here.
668          */
669         if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
670                 ZFS_EXIT(zfsvfs);
671                 return (SET_ERROR(EROFS));
672         }

674         /*
675          * If immutable or not appending then return EPERM
676          */
677         if ((zp->z_pflags & (ZFS_IMMUTABLE | ZFS_READONLY)) ||
678             ((zp->z_pflags & ZFS_APPENDONLY) && !(ioflag & FAPPEND) &&
679             (uio->uio_loffset < zp->z_size))) {
680                 ZFS_EXIT(zfsvfs);
681                 return (SET_ERROR(EPERM));
682         }

684         zilog = zfsvfs->z_log;

686         /*
687          * Validate file offset
688          */
689         woff = ioflag & FAPPEND ? zp->z_size : uio->uio_loffset;
690         if (woff < 0) {
691                 ZFS_EXIT(zfsvfs);
692                 return (SET_ERROR(EINVAL));
693         }

695         /*
696          * Check for mandatory locks before calling zfs_range_lock()
697          * in order to prevent a deadlock with locks set via fcntl().
698          */
699         if (MANDMODE((mode_t)zp->z_mode) &&
700             (error = chklock(vp, FWRITE, woff, n, uio->uio_fmode, ct)) != 0) {
701                 ZFS_EXIT(zfsvfs);
702                 return (error);
703         }

705         /*
706          * Pre-fault the pages to ensure slow (eg NFS) pages
707          * don't hold up txg.
708          * Skip this if uio contains loaned arc_buf.
709          */
710         if ((uio->uio_extflg == UIO_XUIO) &&
711             (((xuio_t *)uio)->xu_type == UIOTYPE_ZEROCOPY))
712                 xuio = (xuio_t *)uio;
713         else
714                 uio_prefaultpages(MIN(n, max_blksz), uio);

716         /*
717          * If in append mode, set the io offset pointer to eof.
718          */
719         if (ioflag & FAPPEND) {
720                 /*
721                  * Obtain an appending range lock to guarantee file append
```



```
722                  * semantics.  We reset the write offset once we have the lock.
723                  */
724                 rl = zfs_range_lock(zp, 0, n, RL_APPEND);
725                 woff = rl->r_off;
726                 if (rl->r_len == UINT64_MAX) {
727                         /*
728                          * We overlocked the file because this write will cause
729                          * the file block size to increase.
730                          * Note that zp_size cannot change with this lock held.
731                          */
732                         woff = zp->z_size;
733                 }
734                 uio->uio_loffset = woff;
735         } else {
736                 /*
737                  * Note that if the file block size will change as a result of
738                  * this write, then this range lock will lock the entire file
739                  * so that we can re-write the block safely.
740                  */
741                 rl = zfs_range_lock(zp, woff, n, RL_WRITER);
742         }

744         if (woff >= limit) {
745                 zfs_range_unlock(rl);
746                 ZFS_EXIT(zfsvfs);
747                 return (SET_ERROR(EFBIG));
748         }

750         if ((woff + n) > limit || woff > (limit - n))
751                 n = limit - woff;

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

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

758         /*
759          * Write the file in reasonable size chunks.  Each chunk is written
760          * in a separate transaction; this keeps the intent log records small
761          * and allows us to do more fine-grained space accounting.
762          */
763         while (n > 0) {
764                 abuf = NULL;
765                 woff = uio->uio_loffset;
766                 if (zfs_owner_overquota(zfsvfs, zp, B_FALSE) ||
767                     zfs_owner_overquota(zfsvfs, zp, B_TRUE)) {
768                         if (abuf != NULL)
769                                 dmu_return_arcbuf(abuf);
770                         error = SET_ERROR(EDQUOT);
771                         break;
772                 }

774                 if (xuio && abuf == NULL) {
775                         ASSERT(i_iov < iovcnt);
776                         aiov = &iovp[i_iov];
777                         abuf = dmu_xuio_arcbuf(xuio, i_iov);
778                         dmu_xuio_clear(xuio, i_iov);
779                         DTRACE_PROBE3(zfs_cp_write, int, i_iov,
780                             iovec_t *, aiov, arc_buf_t *, abuf);
781                         ASSERT((aiov->iov_base == abuf->b_data) ||
782                             ((char *)aiov->iov_base - (char *)abuf->b_data +
783                             aiov->iov_len == arc_buf_size(abuf)));
784                         i_iov++;
785                 } else if (abuf == NULL && n >= max_blksz &&
786                     woff >= zp->z_size &&
787                     P2PHASE(woff, max_blksz) == 0 &&
```

```
788                     zp->z_blksz == max_blksz) {
789                         /*
790                          * This write covers a full block.  "Borrow" a buffer
791                          * from the dmu so that we can fill it before we enter
792                          * a transaction.  This avoids the possibility of
793                          * holding up the transaction if the data copy hangs
794                          * up on a pagefault (e.g., from an NFS server mapping).
795                          */
796                         size_t cbytes;

798                         abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
799                             max_blksz);
800                         ASSERT(abuf != NULL);
801                         ASSERT(arc_buf_size(abuf) == max_blksz);
802                         if (error = uiocopy(abuf->b_data, max_blksz,
803                             UIO_WRITE, uio, &cbytes)) {
804                             dmu_return_arcbuf(abuf);
805                             break;
806                         }
807                         ASSERT(cbytes == max_blksz);
808                 }

810                 /*
811                  * Start a transaction.
812                  */
813                 tx = dmu_tx_create(zfsvfs->z_os);
814                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
815                 dmu_tx_hold_write(tx, zp->z_id, woff, MIN(n, max_blksz));
816                 zfs_sa_upgrade_txholds(tx, zp);
817                 error = dmu_tx_assign(tx, TXG_WAIT);
818                 if (error) {
819                         dmu_tx_abort(tx);
820                         if (abuf != NULL)
821                                 dmu_return_arcbuf(abuf);
822                         break;
823                 }

825                 /*
826                  * If zfs_range_lock() over-locked we grow the blocksize
827                  * and then reduce the lock range.  This will only happen
828                  * on the first iteration since zfs_range_reduce() will
829                  * shrink down r_len to the appropriate size.
830                  */
831                 if (rl->r_len == UINT64_MAX) {
832                         uint64_t new_blksz;

834                         if (zp->z_blksz > max_blksz) {
835                                 ASSERT(!ISP2(zp->z_blksz));
836                                 new_blksz = MIN(end_size, SPA_MAXBLOCKSIZE);
837                         } else {
838                                 new_blksz = MIN(end_size, max_blksz);
839                         }
840                         zfs_grow_blocksize(zp, new_blksz, tx);
841                         zfs_range_reduce(rl, woff, n);
842                 }

844                 /*
845                  * XXX - should we really limit each write to z_max_blksz?
846                  * Perhaps we should use SPA_MAXBLOCKSIZE chunks?
847                  */
848                 nbytes = MIN(n, max_blksz - P2PHASE(woff, max_blksz));

850                 if (abuf == NULL) {
851                         tx_bytes = uio->uio_resid;
852                         error = dmu_write_uio_dbuf(sa_get_db(zp->z_sa_hdl),
853                             uio, nbytes, tx);
```



```
854                         tx_bytes -= uio->uio_resid;
855                 } else {
856                         tx_bytes = nbytes;
857                         ASSERT(xuio == NULL || tx_bytes == aiov->iov_len);
858                         /*
859                          * If this is not a full block write, but we are
860                          * extending the file past EOF and this data starts
861                          * block-aligned, use assign_arcbuf().  Otherwise,
862                          * write via dmu_write().
863                          */
864                         if (tx_bytes < max_blksz && (!write_eof ||
865                             aiov->iov_base != abuf->b_data)) {
866                                 ASSERT(xuio);
867                                 dmu_write(zfsvfs->z_os, zp->z_id, woff,
868                                     aiov->iov_len, aiov->iov_base, tx);
869                                 dmu_return_arcbuf(abuf);
870                                 xuio_stat_wbuf_copied();
871                         } else {
872                                 ASSERT(xuio || tx_bytes == max_blksz);
873                                 dmu_assign_arcbuf(sa_get_db(zp->z_sa_hdl),
874                                     woff, abuf, tx);
875                         }
876                         ASSERT(tx_bytes <= uio->uio_resid);
877                         uioskip(uio, tx_bytes);
878                 }
879                 if (tx_bytes && vn_has_cached_data(vp)) {
880                         update_pages(vp, woff,
881                             tx_bytes, zfsvfs->z_os, zp->z_id);
882                 }

884                 /*
885                  * If we made no progress, we're done.  If we made even
886                  * partial progress, update the znode and ZIL accordingly.
887                  */
888                 if (tx_bytes == 0) {
889                         (void) sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zfsvfs),
890                             (void *)&zp->z_size, sizeof (uint64_t), tx);
891                         dmu_tx_commit(tx);
892                         ASSERT(error != 0);
893                         break;
894                 }

896                 /*
897                  * Clear Set-UID/Set-GID bits on successful write if not
898                  * privileged and at least one of the excute bits is set.
899                  *
900                  * It would be nice to to this after all writes have
901                  * been done, but that would still expose the ISUID/ISGID
902                  * to another app after the partial write is committed.
903                  *
904                  * Note: we don't call zfs_fuid_map_id() here because
905                  * user 0 is not an ephemeral uid.
906                  */
907                 mutex_enter(&zp->z_acl_lock);
908                 if ((zp->z_mode & (S_IXUSR | (S_IXUSR >> 3) |
909                     (S_IXUSR >> 6))) != 0 &&
910                     (zp->z_mode & (S_ISUID | S_ISGID)) != 0 &&
911                     secpolicy_vnode_setid_retain(cr,
912                     (zp->z_mode & S_ISUID) != 0 && zp->z_uid == 0) != 0) {
913                         uint64_t newmode;
914                         zp->z_mode &= ~(S_ISUID | S_ISGID);
915                         newmode = zp->z_mode;
916                         (void) sa_update(zp->z_sa_hdl, SA_ZPL_MODE(zfsvfs),
917                             (void *)&newmode, sizeof (uint64_t), tx);
918                 }
919                 mutex_exit(&zp->z_acl_lock);
```

```
921                 zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
922                     B_TRUE);
923
924                 /*
925                  * Update the file size (zp_size) if it has changed;
926                  * account for possible concurrent updates.
927                  */
928                 while ((end_size = zp->z_size) < uio->uio_loffset) {
929                         (void) atomic_cas_64(&zp->z_size, end_size,
930                             uio->uio_loffset);
931                         ASSERT(error == 0);
932                 }
933                 /*
934                  * If we are replaying and eof is non zero then force
935                  * the file size to the specified eof. Note, there's no
936                  * concurrency during replay.
937                  */
938                 if (zfsvfs->z_replay && zfsvfs->z_replay_eof != 0)
939                         zp->z_size = zfsvfs->z_replay_eof;
940
941                 error = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
942
943                 zfs_log_write(zilog, tx, TX_WRITE, zp, woff, tx_bytes, ioflag);
944                 dmu_tx_commit(tx);
945
946                 if (error != 0)
947                         break;
948                 ASSERT(tx_bytes == nbytes);
949                 n -= nbytes;
950
951                 if (!xuio && n > 0)
952                         uio_prefaultpages(MIN(n, max_blksz), uio);
953         }
954
955         zfs_range_unlock(rl);
956
957         /*
958          * If we're in replay mode, or we made no progress, return error.
959          * Otherwise, it's at least a partial write, so it's successful.
960          */
961         if (zfsvfs->z_replay || uio->uio_resid == start_resid) {
962                 ZFS_EXIT(zfsvfs);
963                 return (error);
964         }
965
966         if (ioflag & (FSYNC | FDSYNC) ||
967             zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
968                 zil_commit(zilog, zp->z_id);
969
970         ZFS_EXIT(zfsvfs);
971         return (0);
972 }
973
974 #define ZFS_RESERVE_CHUNK (2 * 1024 * 1024)
975 /* ARGSUSED */
976 static int
977 zfs_zero_write(vnode_t *vp, uint64_t size, cred_t *cr, caller_context_t *ct)
978 {
979         znode_t         *zp = VTOZ(vp);
980         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
981         int             count = 0;
982         sa_bulk_attr_t  bulk[4];
983         uint64_t        mtime[2], ctime[2];
984         rl_t            *rl;
985         int             error = 0;
```



```
986         dmu_tx_t        *tx = NULL;
987         uint64_t        end_size;
988         uint64_t        pos = 0;
989
990         if (zp->z_size > 0)
991                 return (EFBIG);
992         if (size == 0)
993                 return (0);
994
995         ZFS_ENTER(zfsvfs);
996         ZFS_VERIFY_ZP(zp);
997
998         if (!spa_feature_is_enabled(zfsvfs->z_os->os_spa,
999             SPA_FEATURE_SPACE_RESERVATION))
1000        {
1001                ZFS_EXIT(zfsvfs);
1002                return (ENOTSUP);
1003        }
1004
1005        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
1006        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);
1007        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_SIZE(zfsvfs), NULL,
1008            &zp->z_size, 8);
1009        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
1010            &zp->z_pflags, 8);
1011
1012        /*
1013         * If immutable or not appending then return EPERM
1014         */
1015        if ((zp->z_pflags & (ZFS_IMMUTABLE | ZFS_READONLY))) {
1016                ZFS_EXIT(zfsvfs);
1017                return (EPERM);
1018        }
1019
1020        rl = zfs_range_lock(zp, 0, size, RL_WRITER);
1021
1022        if (zfs_owner_overquota(zfsvfs, zp, B_FALSE) ||
1023            zfs_owner_overquota(zfsvfs, zp, B_TRUE)) {
1024                error = EDQUOT;
1025                goto out;
1026        }
1027
1028        while (pos < size) {
1029                uint64_t length = size - pos;
1030                length = MIN(length, ZFS_RESERVE_CHUNK);
1031 again:
1032                tx = dmu_tx_create(zfsvfs->z_os);
1033                dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1034                dmu_tx_hold_write(tx, zp->z_id, pos, length);
1035                zfs_sa_upgrade_txholds(tx, zp);
1036                error = dmu_tx_assign(tx, TXG_NOWAIT);
1037                if (error) {
1038                        if (error == ERESTART) {
1039                                dmu_tx_wait(tx);
1040                                dmu_tx_abort(tx);
1041                                goto again;
1042                        }
1043                        dmu_tx_abort(tx);
1044                        goto out;
1045                }
1046
1047                if (pos == 0)
1048                        zfs_grow_blocksize(zp, MIN(size, zfsvfs->z_max_blksz), t
1049                dmu_write_zero(zfsvfs->z_os, zp->z_id, pos, length, tx);
1050
1051                zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime, B_TR
```

```
1053                 pos += length;
1054                 while ((end_size = zp->z_size) < pos)
1055                         (void) atomic_cas_64(&zp->z_size, end_size, pos);

1057                 error = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);

1059                 dmu_tx_commit(tx);
1060                 if (error)
1061                         goto out;
1062         }
1063 out:
1064         zfs_range_unlock(rl);
1065         ZFS_EXIT(zfsvfs);

1067         return (error);
1068 }

1070 #endif /* ! codereview */
1071 void
1072 zfs_get_done(zgd_t *zgd, int error)
1073 {
1074         znode_t *zp = zgd->zgd_private;
1075         objset_t *os = zp->z_zfsvfs->z_os;

1077         if (zgd->zgd_db)
1078                 dmu_buf_rele(zgd->zgd_db, zgd);

1080         zfs_range_unlock(zgd->zgd_rl);

1082         /*
1083          * Release the vnode asynchronously as we currently have the
1084          * txg stopped from syncing.
1085          */
1086         VN_RELE_ASYNC(ZTOV(zp), dsl_pool_vnrele_taskq(dmu_objset_pool(os)));

1088         if (error == 0 && zgd->zgd_bp)
1089                 zil_add_block(zgd->zgd_zilog, zgd->zgd_bp);

1091         kmem_free(zgd, sizeof (zgd_t));
1092 }

1094 #ifdef DEBUG
1095 static int zil_fault_io = 0;
1096 #endif

1098 /*
1099  * Get data to generate a TX_WRITE intent log record.
1100  */
1101 int
1102 zfs_get_data(void *arg, lr_write_t *lr, char *buf, zio_t *zio)
1103 {
1104         zfsvfs_t *zfsvfs = arg;
1105         objset_t *os = zfsvfs->z_os;
1106         znode_t *zp;
1107         uint64_t object = lr->lr_foid;
1108         uint64_t offset = lr->lr_offset;
1109         uint64_t size = lr->lr_length;
1110         blkptr_t *bp = &lr->lr_blkptr;
1111         dmu_buf_t *db;
1112         zgd_t *zgd;
1113         int error = 0;

1115         ASSERT(zio != NULL);
1116         ASSERT(size != 0);
```



```
1118         /*
1119          * Nothing to do if the file has been removed
1120          */
1121         if (zfs_zget(zfsvfs, object, &zp) != 0)
1122                 return (SET_ERROR(ENOENT));
1123         if (zp->z_unlinked) {
1124                 /*
1125                  * Release the vnode asynchronously as we currently have the
1126                  * txg stopped from syncing.
1127                  */
1128                 VN_RELE_ASYNC(ZTOV(zp),
1129                     dsl_pool_vnrele_taskq(dmu_objset_pool(os)));
1130                 return (SET_ERROR(ENOENT));
1131         }

1133         zgd = (zgd_t *)kmem_zalloc(sizeof (zgd_t), KM_SLEEP);
1134         zgd->zgd_zilog = zfsvfs->z_log;
1135         zgd->zgd_private = zp;

1137         /*
1138          * Write records come in two flavors: immediate and indirect.
1139          * For small writes it's cheaper to store the data with the
1140          * log record (immediate); for large writes it's cheaper to
1141          * sync the data and get a pointer to it (indirect) so that
1142          * we don't have to write the data twice.
1143          */
1144         if (buf != NULL) { /* immediate write */
1145                 zgd->zgd_rl = zfs_range_lock(zp, offset, size, RL_READER);
1146                 /* test for truncation needs to be done while range locked */
1147                 if (offset >= zp->z_size) {
1148                         error = SET_ERROR(ENOENT);
1149                 } else {
1150                         error = dmu_read(os, object, offset, size, buf,
1151                             DMU_READ_NO_PREFETCH);
1152                 }
1153                 ASSERT(error == 0 || error == ENOENT);
1154         } else { /* indirect write */
1155                 /*
1156                  * Have to lock the whole block to ensure when it's
1157                  * written out and it's checksum is being calculated
1158                  * that no one can change the data. We need to re-check
1159                  * blocksize after we get the lock in case it's changed!
1160                  */
1161                 for (;;) {
1162                         uint64_t blkoff;
1163                         size = zp->z_blksz;
1164                         blkoff = ISP2(size) ? P2PHASE(offset, size) : offset;
1165                         offset -= blkoff;
1166                         zgd->zgd_rl = zfs_range_lock(zp, offset, size,
1167                             RL_READER);
1168                         if (zp->z_blksz == size)
1169                                 break;
1170                         offset += blkoff;
1171                         zfs_range_unlock(zgd->zgd_rl);
1172                 }
1173                 /* test for truncation needs to be done while range locked */
1174                 if (lr->lr_offset >= zp->z_size)
1175                         error = SET_ERROR(ENOENT);
1176 #ifdef DEBUG
1177                 if (zil_fault_io) {
1178                         error = SET_ERROR(EIO);
1179                         zil_fault_io = 0;
1180                 }
1181 #endif
1182                 if (error == 0)
1183                         error = dmu_buf_hold(os, object, offset, zgd, &db,
```

```
1184                            DMU_READ_NO_PREFETCH);

1186                if (error == 0) {
1187                        blkptr_t *obp = dmu_buf_get_blkptr(db);
1188                        if (obp) {
1189                                ASSERT(BP_IS_HOLE(bp));
1190                                *bp = *obp;
1191                        }

1193                        zgd->zgd_db = db;
1194                        zgd->zgd_bp = bp;

1196                        ASSERT(db->db_offset == offset);
1197                        ASSERT(db->db_size == size);

1199                        error = dmu_sync(zio, lr->lr_common.lrc_txg,
1200                            zfs_get_done, zgd);
1201                        ASSERT(error || lr->lr_length <= zp->z_blksz);

1203                        /*
1204                         * On success, we need to wait for the write I/O
1205                         * initiated by dmu_sync() to complete before we can
1206                         * release this dbuf.  We will finish everything up
1207                         * in the zfs_get_done() callback.
1208                         */
1209                        if (error == 0)
1210                                return (0);

1212                        if (error == EALREADY) {
1213                                lr->lr_common.lrc_txtype = TX_WRITE2;
1214                                error = 0;
1215                        }
1216                }
1217        }

1219        zfs_get_done(zgd, error);

1221        return (error);
1222 }

1224 /*ARGSUSED*/
1225 static int
1226 zfs_access(vnode_t *vp, int mode, int flag, cred_t *cr,
1227     caller_context_t *ct)
1228 {
1229        znode_t *zp = VTOZ(vp);
1230        zfsvfs_t *zfsvfs = zp->z_zfsvfs;
1231        int error;

1233        ZFS_ENTER(zfsvfs);
1234        ZFS_VERIFY_ZP(zp);

1236        if (flag & V_ACE_MASK)
1237                error = zfs_zaccess(zp, mode, flag, B_FALSE, cr);
1238        else
1239                error = zfs_zaccess_rwx(zp, mode, flag, cr);

1241        ZFS_EXIT(zfsvfs);
1242        return (error);
1243 }

1245 /*
1246  * If vnode is for a device return a specfs vnode instead.
1247  */
1248 static int
1249 specvp_check(vnode_t **vpp, cred_t *cr)
```



```
1250 {
1251        int error = 0;

1253        if (IS_DEVVP(*vpp)) {
1254                struct vnode *svp;

1256                svp = specvp(*vpp, (*vpp)->v_rdev, (*vpp)->v_type, cr);
1257                VN_RELE(*vpp);
1258                if (svp == NULL)
1259                        error = SET_ERROR(ENOSYS);
1260                *vpp = svp;
1261        }
1262        return (error);
1263 }


1266 /*
1267  * Lookup an entry in a directory, or an extended attribute directory.
1268  * If it exists, return a held vnode reference for it.
1269  *
1270  *      IN:     dvp     - vnode of directory to search.
1271  *              nm      - name of entry to lookup.
1272  *              pnp     - full pathname to lookup [UNUSED].
1273  *              flags   - LOOKUP_XATTR set if looking for an attribute.
1274  *              rdir    - root directory vnode [UNUSED].
1275  *              cr      - credentials of caller.
1276  *              ct      - caller context
1277  *              direntflags - directory lookup flags
1278  *              realpnp - returned pathname.
1279  *
1280  *      OUT:    vpp     - vnode of located entry, NULL if not found.
1281  *
1282  *      RETURN: 0 on success, error code on failure.
1283  *
1284  * Timestamps:
1285  *      NA
1286  */
1287 /* ARGSUSED */
1288 static int
1289 zfs_lookup(vnode_t *dvp, char *nm, vnode_t **vpp, struct pathname *pnp,
1290     int flags, vnode_t *rdir, cred_t *cr,  caller_context_t *ct,
1291     int *direntflags, pathname_t *realpnp)
1292 {
1293        znode_t *zdp = VTOZ(dvp);
1294        zfsvfs_t *zfsvfs = zdp->z_zfsvfs;
1295        int     error = 0;

1297        /* fast path */
1298        if (!(flags & (LOOKUP_XATTR | FIGNORECASE))) {

1300                if (dvp->v_type != VDIR) {
1301                        return (SET_ERROR(ENOTDIR));
1302                } else if (zdp->z_sa_hdl == NULL) {
1303                        return (SET_ERROR(EIO));
1304                }

1306                if (nm[0] == 0 || (nm[0] == '.' && nm[1] == '\0')) {
1307                        error = zfs_fastaccesschk_execute(zdp, cr);
1308                        if (!error) {
1309                                *vpp = dvp;
1310                                VN_HOLD(*vpp);
1311                                return (0);
1312                        }
1313                        return (error);
1314                } else {
1315                        vnode_t *tvp = dnlc_lookup(dvp, nm);
```

```
1317                        if (tvp) {
1318                                error = zfs_fastaccesschk_execute(zdp, cr);
1319                                if (error) {
1320                                        VN_RELE(tvp);
1321                                        return (error);
1322                                }
1323                                if (tvp == DNLC_NO_VNODE) {
1324                                        VN_RELE(tvp);
1325                                        return (SET_ERROR(ENOENT));
1326                                } else {
1327                                        *vpp = tvp;
1328                                        return (specvp_check(vpp, cr));
1329                                }
1330                        }
1331                }
1332        }

1334        DTRACE_PROBE2(zfs__fastpath__lookup__miss, vnode_t *, dvp, char *, nm);

1336        ZFS_ENTER(zfsvfs);
1337        ZFS_VERIFY_ZP(zdp);

1339        *vpp = NULL;

1341        if (flags & LOOKUP_XATTR) {
1342                /*
1343                 * If the xattr property is off, refuse the lookup request.
1344                 */
1345                if (!(zfsvfs->z_vfs->vfs_flag & VFS_XATTR)) {
1346                        ZFS_EXIT(zfsvfs);
1347                        return (SET_ERROR(EINVAL));
1348                }

1350                /*
1351                 * We don't allow recursive attributes..
1352                 * Maybe someday we will.
1353                 */
1354                if (zdp->z_pflags & ZFS_XATTR) {
1355                        ZFS_EXIT(zfsvfs);
1356                        return (SET_ERROR(EINVAL));
1357                }

1359                if (error = zfs_get_xattrdir(VTOZ(dvp), vpp, cr, flags)) {
1360                        ZFS_EXIT(zfsvfs);
1361                        return (error);
1362                }

1364                /*
1365                 * Do we have permission to get into attribute directory?
1366                 */

1368                if (error = zfs_zaccess(VTOZ(*vpp), ACE_EXECUTE, 0,
1369                    B_FALSE, cr)) {
1370                        VN_RELE(*vpp);
1371                        *vpp = NULL;
1372                }

1374                ZFS_EXIT(zfsvfs);
1375                return (error);
1376        }

1378        if (dvp->v_type != VDIR) {
1379                ZFS_EXIT(zfsvfs);
1380                return (SET_ERROR(ENOTDIR));
1381        }
```



```
1383        /*
1384         * Check accessibility of directory.
1385         */

1387        if (error = zfs_zaccess(zdp, ACE_EXECUTE, 0, B_FALSE, cr)) {
1388                ZFS_EXIT(zfsvfs);
1389                return (error);
1390        }

1392        if (zfsvfs->z_utf8 && u8_validate(nm, strlen(nm),
1393            NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1394                ZFS_EXIT(zfsvfs);
1395                return (SET_ERROR(EILSEQ));
1396        }

1398        error = zfs_dirlook(zdp, nm, vpp, flags, direntflags, realpnp);
1399        if (error == 0)
1400                error = specvp_check(vpp, cr);

1402        ZFS_EXIT(zfsvfs);
1403        return (error);
1404 }

1406 /*
1407  * Attempt to create a new entry in a directory.  If the entry
1408  * already exists, truncate the file if permissible, else return
1409  * an error.  Return the vp of the created or trunc'd file.
1410  *
1411  *      IN:     dvp     - vnode of directory to put new file entry in.
1412  *              name    - name of new file entry.
1413  *              vap     - attributes of new file.
1414  *              excl    - flag indicating exclusive or non-exclusive mode.
1415  *              mode    - mode to open file with.
1416  *              cr      - credentials of caller.
1417  *              flag    - large file flag [UNUSED].
1418  *              ct      - caller context
1419  *              vsecp   - ACL to be set
1420  *
1421  *      OUT:    vpp     - vnode of created or trunc'd entry.
1422  *
1423  *      RETURN: 0 on success, error code on failure.
1424  *
1425  * Timestamps:
1426  *      dvp - ctime|mtime updated if new entry created
1427  *       vp - ctime|mtime always, atime if new
1428  */

1430 /* ARGSUSED */
1431 static int
1432 zfs_create(vnode_t *dvp, char *name, vattr_t *vap, vcexcl_t excl,
1433     int mode, vnode_t **vpp, cred_t *cr, int flag, caller_context_t *ct,
1434     vsecattr_t *vsecp)
1435 {
1436        znode_t         *zp, *dzp = VTOZ(dvp);
1437        zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1438        zilog_t         *zilog;
1439        objset_t        *os;
1440        zfs_dirlock_t   *dl;
1441        dmu_tx_t        *tx;
1442        int             error;
1443        ksid_t          *ksid;
1444        uid_t           uid;
1445        gid_t           gid = crgetgid(cr);
1446        zfs_acl_ids_t   acl_ids;
1447        boolean_t       fuid_dirtied;
```

```
1448         boolean_t       have_acl = B_FALSE;
1449         boolean_t       waited = B_FALSE;

1451         /*
1452          * If we have an ephemeral id, ACL, or XVATTR then
1453          * make sure file system is at proper version
1454          */

1456         ksid = crgetsid(cr, KSID_OWNER);
1457         if (ksid)
1458                 uid = ksid_getid(ksid);
1459         else
1460                 uid = crgetuid(cr);

1462         if (zfsvfs->z_use_fuids == B_FALSE &&
1463             (vsecp || (vap->va_mask & AT_XVATTR) ||
1464             IS_EPHEMERAL(uid) || IS_EPHEMERAL(gid)))
1465                 return (SET_ERROR(EINVAL));

1467         ZFS_ENTER(zfsvfs);
1468         ZFS_VERIFY_ZP(dzp);
1469         os = zfsvfs->z_os;
1470         zilog = zfsvfs->z_log;

1472         if (zfsvfs->z_utf8 && u8_validate(name, strlen(name),
1473             NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1474                 ZFS_EXIT(zfsvfs);
1475                 return (SET_ERROR(EILSEQ));
1476         }

1478         if (vap->va_mask & AT_XVATTR) {
1479                 if ((error = secpolicy_xvattr((xvattr_t *)vap,
1480                     crgetuid(cr), cr, vap->va_type)) != 0) {
1481                         ZFS_EXIT(zfsvfs);
1482                         return (error);
1483                 }
1484         }
1485 top:
1486         *vpp = NULL;

1488         if ((vap->va_mode & VSVTX) && secpolicy_vnode_stky_modify(cr))
1489                 vap->va_mode &= ~VSVTX;

1491         if (*name == '\0') {
1492                 /*
1493                  * Null component name refers to the directory itself.
1494                  */
1495                 VN_HOLD(dvp);
1496                 zp = dzp;
1497                 dl = NULL;
1498                 error = 0;
1499         } else {
1500                 /* possible VN_HOLD(zp) */
1501                 int zflg = 0;

1503                 if (flag & FIGNORECASE)
1504                         zflg |= ZCILOOK;

1506                 error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1507                     NULL, NULL);
1508                 if (error) {
1509                         if (have_acl)
1510                                 zfs_acl_ids_free(&acl_ids);
1511                         if (strcmp(name, "..") == 0)
1512                                 error = SET_ERROR(EISDIR);
1513                         ZFS_EXIT(zfsvfs);
```



```
1514                         return (error);
1515                 }
1516         }

1518         if (zp == NULL) {
1519                 uint64_t txtype;

1521                 /*
1522                  * Create a new file object and update the directory
1523                  * to reference it.
1524                  */
1525                 if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
1526                         if (have_acl)
1527                                 zfs_acl_ids_free(&acl_ids);
1528                         goto out;
1529                 }

1531                 /*
1532                  * We only support the creation of regular files in
1533                  * extended attribute directories.
1534                  */

1536                 if ((dzp->z_pflags & ZFS_XATTR) &&
1537                     (vap->va_type != VREG)) {
1538                         if (have_acl)
1539                                 zfs_acl_ids_free(&acl_ids);
1540                         error = SET_ERROR(EINVAL);
1541                         goto out;
1542                 }

1544                 if (!have_acl && (error = zfs_acl_ids_create(dzp, 0, vap,
1545                     cr, vsecp, &acl_ids)) != 0)
1546                         goto out;
1547                 have_acl = B_TRUE;

1549                 if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
1550                         zfs_acl_ids_free(&acl_ids);
1551                         error = SET_ERROR(EDQUOT);
1552                         goto out;
1553                 }

1555                 tx = dmu_tx_create(os);

1557                 dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
1558                     ZFS_SA_BASE_ATTR_SIZE);

1560                 fuid_dirtied = zfsvfs->z_fuid_dirty;
1561                 if (fuid_dirtied)
1562                         zfs_fuid_txhold(zfsvfs, tx);
1563                 dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
1564                 dmu_tx_hold_sa(tx, dzp->z_sa_hdl, B_FALSE);
1565                 if (!zfsvfs->z_use_sa &&
1566                     acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
1567                         dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
1568                             0, acl_ids.z_aclp->z_acl_bytes);
1569                 }
1570                 error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
1571                 if (error) {
1572                         zfs_dirent_unlock(dl);
1573                         if (error == ERESTART) {
1574                                 waited = B_TRUE;
1575                                 dmu_tx_wait(tx);
1576                                 dmu_tx_abort(tx);
1577                                 goto top;
1578                         }
1579                         zfs_acl_ids_free(&acl_ids);
```

```
1580                        dmu_tx_abort(tx);
1581                        ZFS_EXIT(zfsvfs);
1582                        return (error);
1583                }
1584                zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

1586                if (fuid_dirtied)
1587                        zfs_fuid_sync(zfsvfs, tx);

1589                (void) zfs_link_create(dl, zp, tx, ZNEW);
1590                txtype = zfs_log_create_txtype(Z_FILE, vsecp, vap);
1591                if (flag & FIGNORECASE)
1592                        txtype |= TX_CI;
1593                zfs_log_create(zilog, tx, txtype, dzp, zp, name,
1594                    vsecp, acl_ids.z_fuidp, vap);
1595                zfs_acl_ids_free(&acl_ids);
1596                dmu_tx_commit(tx);
1597        } else {
1598                int aflags = (flag & FAPPEND) ? V_APPEND : 0;

1600                if (have_acl)
1601                        zfs_acl_ids_free(&acl_ids);
1602                have_acl = B_FALSE;

1604                /*
1605                 * A directory entry already exists for this name.
1606                 */
1607                /*
1608                 * Can't truncate an existing file if in exclusive mode.
1609                 */
1610                if (excl == EXCL) {
1611                        error = SET_ERROR(EEXIST);
1612                        goto out;
1613                }
1614                /*
1615                 * Can't open a directory for writing.
1616                 */
1617                if ((ZTOV(zp)->v_type == VDIR) && (mode & S_IWRITE)) {
1618                        error = SET_ERROR(EISDIR);
1619                        goto out;
1620                }
1621                /*
1622                 * Verify requested access to file.
1623                 */
1624                if (mode && (error = zfs_zaccess_rwx(zp, mode, aflags, cr))) {
1625                        goto out;
1626                }

1628                mutex_enter(&dzp->z_lock);
1629                dzp->z_seq++;
1630                mutex_exit(&dzp->z_lock);

1632                /*
1633                 * Truncate regular files if requested.
1634                 */
1635                if ((ZTOV(zp)->v_type == VREG) &&
1636                    (vap->va_mask & AT_SIZE) && (vap->va_size == 0)) {
1637                        /* we can't hold any locks when calling zfs_freesp() */
1638                        zfs_dirent_unlock(dl);
1639                        dl = NULL;
1640                        error = zfs_freesp(zp, 0, 0, mode, TRUE);
1641                        if (error == 0) {
1642                                vnevent_create(ZTOV(zp), ct);
1643                        }
1644                }
1645        }
```



```
1646 out:

1648        if (dl)
1649                zfs_dirent_unlock(dl);

1651        if (error) {
1652                if (zp)
1653                        VN_RELE(ZTOV(zp));
1654        } else {
1655                *vpp = ZTOV(zp);
1656                error = specvp_check(vpp, cr);
1657        }

1659        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1660                zil_commit(zilog, 0);

1662        ZFS_EXIT(zfsvfs);
1663        return (error);
1664 }

1666 /*
1667  * Remove an entry from a directory.
1668  *
1669  *      IN:     dvp     - vnode of directory to remove entry from.
1670  *              name    - name of entry to remove.
1671  *              cr      - credentials of caller.
1672  *              ct      - caller context
1673  *              flags   - case flags
1674  *
1675  *      RETURN: 0 on success, error code on failure.
1676  *
1677  * Timestamps:
1678  *      dvp - ctime|mtime
1679  *       vp - ctime (if nlink > 0)
1680  */

1682 uint64_t null_xattr = 0;

1684 /*ARGSUSED*/
1685 static int
1686 zfs_remove(vnode_t *dvp, char *name, cred_t *cr, caller_context_t *ct,
1687     int flags)
1688 {
1689        znode_t         *zp, *dzp = VTOZ(dvp);
1690        znode_t         *xzp;
1691        vnode_t         *vp;
1692        zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1693        zilog_t         *zilog;
1694        uint64_t        acl_obj, xattr_obj;
1695        uint64_t        xattr_obj_unlinked = 0;
1696        uint64_t        obj = 0;
1697        zfs_dirlock_t   *dl;
1698        dmu_tx_t        *tx;
1699        boolean_t       may_delete_now, delete_now = FALSE;
1700        boolean_t       unlinked, toobig = FALSE;
1701        uint64_t        txtype;
1702        pathname_t      *realnmp = NULL;
1703        pathname_t      realnm;
1704        int             error;
1705        int             zflg = ZEXISTS;
1706        boolean_t       waited = B_FALSE;

1708        ZFS_ENTER(zfsvfs);
1709        ZFS_VERIFY_ZP(dzp);
1710        zilog = zfsvfs->z_log;
```

```
1712         if (flags & FIGNORECASE) {
1713                 zflg |= ZCILOOK;
1714                 pn_alloc(&realnm);
1715                 realnmp = &realnm;
1716         }

1718 top:
1719         xattr_obj = 0;
1720         xzp = NULL;
1721         /*
1722          * Attempt to lock directory; fail if entry doesn't exist.
1723          */
1724         if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1725             NULL, realnmp)) {
1726                 if (realnmp)
1727                         pn_free(realnmp);
1728                 ZFS_EXIT(zfsvfs);
1729                 return (error);
1730         }

1732         vp = ZTOV(zp);

1734         if (error = zfs_zaccess_delete(dzp, zp, cr)) {
1735                 goto out;
1736         }

1738         /*
1739          * Need to use rmdir for removing directories.
1740          */
1741         if (vp->v_type == VDIR) {
1742                 error = SET_ERROR(EPERM);
1743                 goto out;
1744         }

1746         vnevent_remove(vp, dvp, name, ct);

1748         if (realnmp)
1749                 dnlc_remove(dvp, realnmp->pn_buf);
1750         else
1751                 dnlc_remove(dvp, name);

1753         mutex_enter(&vp->v_lock);
1754         may_delete_now = vp->v_count == 1 && !vn_has_cached_data(vp);
1755         mutex_exit(&vp->v_lock);

1757         /*
1758          * We may delete the znode now, or we may put it in the unlinked set;
1759          * it depends on whether we're the last link, and on whether there are
1760          * other holds on the vnode.  So we dmu_tx_hold() the right things to
1761          * allow for either case.
1762          */
1763         obj = zp->z_id;
1764         tx = dmu_tx_create(zfsvfs->z_os);
1765         dmu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
1766         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1767         zfs_sa_upgrade_txholds(tx, zp);
1768         zfs_sa_upgrade_txholds(tx, dzp);
1769         if (may_delete_now) {
1770                 toobig =
1771                     zp->z_size > zp->z_blksz * DMU_MAX_DELETEBLKCNT;
1772                 /* if the file is too big, only hold_free a token amount */
1773                 dmu_tx_hold_free(tx, zp->z_id, 0,
1774                     (toobig ? DMU_MAX_ACCESS : DMU_OBJECT_END));
1775         }

1777         /* are there any extended attributes? */
```



```
1778         error = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1779             &xattr_obj, sizeof (xattr_obj));
1780         if (error == 0 && xattr_obj) {
1781                 error = zfs_zget(zfsvfs, xattr_obj, &xzp);
1782                 ASSERT0(error);
1783                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
1784                 dmu_tx_hold_sa(tx, xzp->z_sa_hdl, B_FALSE);
1785         }

1787         mutex_enter(&zp->z_lock);
1788         if ((acl_obj = zfs_external_acl(zp)) != 0 && may_delete_now)
1789                 dmu_tx_hold_free(tx, acl_obj, 0, DMU_OBJECT_END);
1790         mutex_exit(&zp->z_lock);

1792         /* charge as an update -- would be nice not to charge at all */
1793         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);

1795         /*
1796          * Mark this transaction as typically resulting in a net free of
1797          * space, unless object removal will be delayed indefinitely
1798          * (due to active holds on the vnode due to the file being open).
1799          */
1800         if (may_delete_now)
1801                 dmu_tx_mark_netfree(tx);

1803         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
1804         if (error) {
1805                 zfs_dirent_unlock(dl);
1806                 VN_RELE(vp);
1807                 if (xzp)
1808                         VN_RELE(ZTOV(xzp));
1809                 if (error == ERESTART) {
1810                         waited = B_TRUE;
1811                         dmu_tx_wait(tx);
1812                         dmu_tx_abort(tx);
1813                         goto top;
1814                 }
1815                 if (realnmp)
1816                         pn_free(realnmp);
1817                 dmu_tx_abort(tx);
1818                 ZFS_EXIT(zfsvfs);
1819                 return (error);
1820         }

1822         /*
1823          * Remove the directory entry.
1824          */
1825         error = zfs_link_destroy(dl, zp, tx, zflg, &unlinked);

1827         if (error) {
1828                 dmu_tx_commit(tx);
1829                 goto out;
1830         }

1832         if (unlinked) {
1833                 /*
1834                  * Hold z_lock so that we can make sure that the ACL obj
1835                  * hasn't changed.  Could have been deleted due to
1836                  * zfs_sa_upgrade().
1837                  */
1838                 mutex_enter(&zp->z_lock);
1839                 mutex_enter(&vp->v_lock);
1840                 (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1841                     &xattr_obj_unlinked, sizeof (xattr_obj_unlinked));
1842                 delete_now = may_delete_now && !toobig &&
1843                     vp->v_count == 1 && !vn_has_cached_data(vp) &&
```

```
1844                    xattr_obj == xattr_obj_unlinked && zfs_external_acl(zp) ==
1845                    acl_obj;
1846                mutex_exit(&vp->v_lock);
1847        }
1848
1849        if (delete_now) {
1850                if (xattr_obj_unlinked) {
1851                        ASSERT3U(xzp->z_links, ==, 2);
1852                        mutex_enter(&xzp->z_lock);
1853                        xzp->z_unlinked = 1;
1854                        xzp->z_links = 0;
1855                        error = sa_update(xzp->z_sa_hdl, SA_ZPL_LINKS(zfsvfs),
1856                            &xzp->z_links, sizeof (xzp->z_links), tx);
1857                        ASSERT3U(error,  ==,  0);
1858                        mutex_exit(&xzp->z_lock);
1859                        zfs_unlinked_add(xzp, tx);
1860
1861                        if (zp->z_is_sa)
1862                                error = sa_remove(zp->z_sa_hdl,
1863                                    SA_ZPL_XATTR(zfsvfs), tx);
1864                        else
1865                                error = sa_update(zp->z_sa_hdl,
1866                                    SA_ZPL_XATTR(zfsvfs), &null_xattr,
1867                                    sizeof (uint64_t), tx);
1868                        ASSERT0(error);
1869                }
1870                mutex_enter(&vp->v_lock);
1871                vp->v_count--;
1872                ASSERT0(vp->v_count);
1873                mutex_exit(&vp->v_lock);
1874                mutex_exit(&zp->z_lock);
1875                zfs_znode_delete(zp, tx);
1876        } else if (unlinked) {
1877                mutex_exit(&zp->z_lock);
1878                zfs_unlinked_add(zp, tx);
1879        }
1880
1881        txtype = TX_REMOVE;
1882        if (flags & FIGNORECASE)
1883                txtype |= TX_CI;
1884        zfs_log_remove(zilog, tx, txtype, dzp, name, obj);
1885
1886        dmu_tx_commit(tx);
1887 out:
1888        if (realnmp)
1889                pn_free(realnmp);
1890
1891        zfs_dirent_unlock(dl);
1892
1893        if (!delete_now)
1894                VN_RELE(vp);
1895        if (xzp)
1896                VN_RELE(ZTOV(xzp));
1897
1898        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1899                zil_commit(zilog, 0);
1900
1901        ZFS_EXIT(zfsvfs);
1902        return (error);
1903 }
1904
1905 /*
1906  * Create a new directory and insert it into dvp using the name
1907  * provided.  Return a pointer to the inserted directory.
1908  *
1909  *      IN:     dvp     - vnode of directory to add subdir to.
```



```
1910  *              dirname - name of new directory.
1911  *              vap     - attributes of new directory.
1912  *              cr      - credentials of caller.
1913  *              ct      - caller context
1914  *              flags   - case flags
1915  *              vsecp   - ACL to be set
1916  *
1917  *      OUT:    vpp     - vnode of created directory.
1918  *
1919  *      RETURN: 0 on success, error code on failure.
1920  *
1921  * Timestamps:
1922  *      dvp - ctime|mtime updated
1923  *       vp - ctime|mtime|atime updated
1924  */
1925 /*ARGSUSED*/
1926 static int
1927 zfs_mkdir(vnode_t *dvp, char *dirname, vattr_t *vap, vnode_t **vpp, cred_t *cr,
1928     caller_context_t *ct, int flags, vsecattr_t *vsecp)
1929 {
1930        znode_t         *zp, *dzp = VTOZ(dvp);
1931        zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
1932        zilog_t         *zilog;
1933        zfs_dirlock_t   *dl;
1934        uint64_t        txtype;
1935        dmu_tx_t        *tx;
1936        int             error;
1937        int             zf = ZNEW;
1938        ksid_t          *ksid;
1939        uid_t           uid;
1940        gid_t           gid = crgetgid(cr);
1941        zfs_acl_ids_t   acl_ids;
1942        boolean_t       fuid_dirtied;
1943        boolean_t       waited = B_FALSE;
1944
1945        ASSERT(vap->va_type == VDIR);
1946
1947        /*
1948         * If we have an ephemeral id, ACL, or XVATTR then
1949         * make sure file system is at proper version
1950         */
1951
1952        ksid = crgetsid(cr, KSID_OWNER);
1953        if (ksid)
1954                uid = ksid_getid(ksid);
1955        else
1956                uid = crgetuid(cr);
1957        if (zfsvfs->z_use_fuids == B_FALSE &&
1958            (vsecp || (vap->va_mask & AT_XVATTR) ||
1959            IS_EPHEMERAL(uid) || IS_EPHEMERAL(gid)))
1960                return (SET_ERROR(EINVAL));
1961
1962        ZFS_ENTER(zfsvfs);
1963        ZFS_VERIFY_ZP(dzp);
1964        zilog = zfsvfs->z_log;
1965
1966        if (dzp->z_pflags & ZFS_XATTR) {
1967                ZFS_EXIT(zfsvfs);
1968                return (SET_ERROR(EINVAL));
1969        }
1970
1971        if (zfsvfs->z_utf8 && u8_validate(dirname,
1972            strlen(dirname), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
1973                ZFS_EXIT(zfsvfs);
1974                return (SET_ERROR(EILSEQ));
1975        }
```

```
1976         if (flags & FIGNORECASE)
1977                 zf |= ZCILOOK;

1979         if (vap->va_mask & AT_XVATTR) {
1980                 if ((error = secpolicy_xvattr((xvattr_t *)vap,
1981                     crgetuid(cr), cr, vap->va_type)) != 0) {
1982                         ZFS_EXIT(zfsvfs);
1983                         return (error);
1984                 }
1985         }

1987         if ((error = zfs_acl_ids_create(dzp, 0, vap, cr,
1988             vsecp, &acl_ids)) != 0) {
1989                 ZFS_EXIT(zfsvfs);
1990                 return (error);
1991         }
1992         /*
1993          * First make sure the new directory doesn't exist.
1994          *
1995          * Existence is checked first to make sure we don't return
1996          * EACCES instead of EEXIST which can cause some applications
1997          * to fail.
1998          */
1999 top:
2000         *vpp = NULL;

2002         if (error = zfs_dirent_lock(&dl, dzp, dirname, &zp, zf,
2003             NULL, NULL)) {
2004                 zfs_acl_ids_free(&acl_ids);
2005                 ZFS_EXIT(zfsvfs);
2006                 return (error);
2007         }

2009         if (error = zfs_zaccess(dzp, ACE_ADD_SUBDIRECTORY, 0, B_FALSE, cr)) {
2010                 zfs_acl_ids_free(&acl_ids);
2011                 zfs_dirent_unlock(dl);
2012                 ZFS_EXIT(zfsvfs);
2013                 return (error);
2014         }

2016         if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
2017                 zfs_acl_ids_free(&acl_ids);
2018                 zfs_dirent_unlock(dl);
2019                 ZFS_EXIT(zfsvfs);
2020                 return (SET_ERROR(EDQUOT));
2021         }

2023         /*
2024          * Add a new entry to the directory.
2025          */
2026         tx = dmu_tx_create(zfsvfs->z_os);
2027         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, dirname);
2028         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
2029         fuid_dirtied = zfsvfs->z_fuid_dirty;
2030         if (fuid_dirtied)
2031                 zfs_fuid_txhold(zfsvfs, tx);
2032         if (!zfsvfs->z_use_sa && acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
2033                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
2034                     acl_ids.z_aclp->z_acl_bytes);
2035         }

2037         dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
2038             ZFS_SA_BASE_ATTR_SIZE);

2040         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
2041         if (error) {
```



```
2042                 zfs_dirent_unlock(dl);
2043                 if (error == ERESTART) {
2044                         waited = B_TRUE;
2045                         dmu_tx_wait(tx);
2046                         dmu_tx_abort(tx);
2047                         goto top;
2048                 }
2049                 zfs_acl_ids_free(&acl_ids);
2050                 dmu_tx_abort(tx);
2051                 ZFS_EXIT(zfsvfs);
2052                 return (error);
2053         }

2055         /*
2056          * Create new node.
2057          */
2058         zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

2060         if (fuid_dirtied)
2061                 zfs_fuid_sync(zfsvfs, tx);

2063         /*
2064          * Now put new name in parent dir.
2065          */
2066         (void) zfs_link_create(dl, zp, tx, ZNEW);

2068         *vpp = ZTOV(zp);

2070         txtype = zfs_log_create_txtype(Z_DIR, vsecp, vap);
2071         if (flags & FIGNORECASE)
2072                 txtype |= TX_CI;
2073         zfs_log_create(zilog, tx, txtype, dzp, zp, dirname, vsecp,
2074             acl_ids.z_fuidp, vap);

2076         zfs_acl_ids_free(&acl_ids);

2078         dmu_tx_commit(tx);

2080         zfs_dirent_unlock(dl);

2082         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
2083                 zil_commit(zilog, 0);

2085         ZFS_EXIT(zfsvfs);
2086         return (0);
2087 }

2089 /*
2090  * Remove a directory subdir entry.  If the current working
2091  * directory is the same as the subdir to be removed, the
2092  * remove will fail.
2093  *
2094  *      IN:     dvp     - vnode of directory to remove from.
2095  *              name    - name of directory to be removed.
2096  *              cwd     - vnode of current working directory.
2097  *              cr      - credentials of caller.
2098  *              ct      - caller context
2099  *              flags   - case flags
2100  *
2101  *      RETURN: 0 on success, error code on failure.
2102  *
2103  * Timestamps:
2104  *      dvp - ctime|mtime updated
2105  */
2106 /*ARGSUSED*/
2107 static int
```

```
2108 zfs_rmdir(vnode_t *dvp, char *name, vnode_t *cwd, cred_t *cr,
2109     caller_context_t *ct, int flags)
2110 {
2111         znode_t         *dzp = VTOZ(dvp);
2112         znode_t         *zp;
2113         vnode_t         *vp;
2114         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
2115         zilog_t         *zilog;
2116         zfs_dirlock_t   *dl;
2117         dmu_tx_t        *tx;
2118         int             error;
2119         int             zflg = ZEXISTS;
2120         boolean_t       waited = B_FALSE;

2122         ZFS_ENTER(zfsvfs);
2123         ZFS_VERIFY_ZP(dzp);
2124         zilog = zfsvfs->z_log;

2126         if (flags & FIGNORECASE)
2127                 zflg |= ZCILOOK;
2128 top:
2129         zp = NULL;

2131         /*
2132          * Attempt to lock directory; fail if entry doesn't exist.
2133          */
2134         if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
2135             NULL, NULL)) {
2136                 ZFS_EXIT(zfsvfs);
2137                 return (error);
2138         }

2140         vp = ZTOV(zp);

2142         if (error = zfs_zaccess_delete(dzp, zp, cr)) {
2143                 goto out;
2144         }

2146         if (vp->v_type != VDIR) {
2147                 error = SET_ERROR(ENOTDIR);
2148                 goto out;
2149         }

2151         if (vp == cwd) {
2152                 error = SET_ERROR(EINVAL);
2153                 goto out;
2154         }

2156         vnevent_rmdir(vp, dvp, name, ct);

2158         /*
2159          * Grab a lock on the directory to make sure that noone is
2160          * trying to add (or lookup) entries while we are removing it.
2161          */
2162         rw_enter(&zp->z_name_lock, RW_WRITER);

2164         /*
2165          * Grab a lock on the parent pointer to make sure we play well
2166          * with the treewalk and directory rename code.
2167          */
2168         rw_enter(&zp->z_parent_lock, RW_WRITER);

2170         tx = dmu_tx_create(zfsvfs->z_os);
2171         dmu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
2172         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
2173         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
```



```
2174         zfs_sa_upgrade_txholds(tx, zp);
2175         zfs_sa_upgrade_txholds(tx, dzp);
2176         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
2177         if (error) {
2178                 rw_exit(&zp->z_parent_lock);
2179                 rw_exit(&zp->z_name_lock);
2180                 zfs_dirent_unlock(dl);
2181                 VN_RELE(vp);
2182                 if (error == ERESTART) {
2183                         waited = B_TRUE;
2184                         dmu_tx_wait(tx);
2185                         dmu_tx_abort(tx);
2186                         goto top;
2187                 }
2188                 dmu_tx_abort(tx);
2189                 ZFS_EXIT(zfsvfs);
2190                 return (error);
2191         }

2193         error = zfs_link_destroy(dl, zp, tx, zflg, NULL);

2195         if (error == 0) {
2196                 uint64_t txtype = TX_RMDIR;
2197                 if (flags & FIGNORECASE)
2198                         txtype |= TX_CI;
2199                 zfs_log_remove(zilog, tx, txtype, dzp, name, ZFS_NO_OBJECT);
2200         }

2202         dmu_tx_commit(tx);

2204         rw_exit(&zp->z_parent_lock);
2205         rw_exit(&zp->z_name_lock);
2206 out:
2207         zfs_dirent_unlock(dl);

2209         VN_RELE(vp);

2211         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
2212                 zil_commit(zilog, 0);

2214         ZFS_EXIT(zfsvfs);
2215         return (error);
2216 }

2218 /*
2219  * Read as many directory entries as will fit into the provided
2220  * buffer from the given directory cursor position (specified in
2221  * the uio structure).
2222  *
2223  *      IN:     vp      - vnode of directory to read.
2224  *              uio     - structure supplying read location, range info,
2225  *                        and return buffer.
2226  *              cr      - credentials of caller.
2227  *              ct      - caller context
2228  *              flags   - case flags
2229  *
2230  *      OUT:    uio     - updated offset and range, buffer filled.
2231  *              eofp    - set to true if end-of-file detected.
2232  *
2233  *      RETURN: 0 on success, error code on failure.
2234  *
2235  * Timestamps:
2236  *      vp - atime updated
2237  *
2238  * Note that the low 4 bits of the cookie returned by zap is always zero.
2239  * This allows us to use the low range for "special" directory entries:
```

```
2240  * We use 0 for '.', and 1 for '..'.  If this is the root of the filesystem,
2241  * we use the offset 2 for the '.zfs' directory.
2242  */
2243 /* ARGSUSED */
2244 static int
2245 zfs_readdir(vnode_t *vp, uio_t *uio, cred_t *cr, int *eofp,
2246     caller_context_t *ct, int flags)
2247 {
2248         znode_t         *zp = VTOZ(vp);
2249         iovec_t         *iovp;
2250         edirent_t       *eodp;
2251         dirent64_t      *odp;
2252         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2253         objset_t        *os;
2254         caddr_t         outbuf;
2255         size_t          bufsize;
2256         zap_cursor_t    zc;
2257         zap_attribute_t zap;
2258         uint_t          bytes_wanted;
2259         uint64_t        offset; /* must be unsigned; checks for < 1 */
2260         uint64_t        parent;
2261         int             local_eof;
2262         int             outcount;
2263         int             error;
2264         uint8_t         prefetch;
2265         boolean_t       check_sysattrs;

2267         ZFS_ENTER(zfsvfs);
2268         ZFS_VERIFY_ZP(zp);

2270         if ((error = sa_lookup(zp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
2271             &parent, sizeof (parent))) != 0) {
2272                 ZFS_EXIT(zfsvfs);
2273                 return (error);
2274         }

2276         /*
2277          * If we are not given an eof variable,
2278          * use a local one.
2279          */
2280         if (eofp == NULL)
2281                 eofp = &local_eof;

2283         /*
2284          * Check for valid iov_len.
2285          */
2286         if (uio->uio_iov->iov_len <= 0) {
2287                 ZFS_EXIT(zfsvfs);
2288                 return (SET_ERROR(EINVAL));
2289         }

2291         /*
2292          * Quit if directory has been removed (posix)
2293          */
2294         if ((*eofp = zp->z_unlinked) != 0) {
2295                 ZFS_EXIT(zfsvfs);
2296                 return (0);
2297         }

2299         error = 0;
2300         os = zfsvfs->z_os;
2301         offset = uio->uio_loffset;
2302         prefetch = zp->z_zn_prefetch;

2304         /*
2305          * Initialize the iterator cursor.
```



```
2306          */
2307         if (offset <= 3) {
2308                 /*
2309                  * Start iteration from the beginning of the directory.
2310                  */
2311                 zap_cursor_init(&zc, os, zp->z_id);
2312         } else {
2313                 /*
2314                  * The offset is a serialized cursor.
2315                  */
2316                 zap_cursor_init_serialized(&zc, os, zp->z_id, offset);
2317         }

2319         /*
2320          * Get space to change directory entries into fs independent format.
2321          */
2322         iovp = uio->uio_iov;
2323         bytes_wanted = iovp->iov_len;
2324         if (uio->uio_segflg != UIO_SYSSPACE || uio->uio_iovcnt != 1) {
2325                 bufsize = bytes_wanted;
2326                 outbuf = kmem_alloc(bufsize, KM_SLEEP);
2327                 odp = (struct dirent64 *)outbuf;
2328         } else {
2329                 bufsize = bytes_wanted;
2330                 outbuf = NULL;
2331                 odp = (struct dirent64 *)iovp->iov_base;
2332         }
2333         eodp = (struct edirent *)odp;

2335         /*
2336          * If this VFS supports the system attribute view interface; and
2337          * we're looking at an extended attribute directory; and we care
2338          * about normalization conflicts on this vfs; then we must check
2339          * for normalization conflicts with the sysattr name space.
2340          */
2341         check_sysattrs = vfs_has_feature(vp->v_vfsp, VFSFT_SYSATTR_VIEWS) &&
2342             (vp->v_flag & V_XATTRDIR) && zfsvfs->z_norm &&
2343             (flags & V_RDDIR_ENTFLAGS);

2345         /*
2346          * Transform to file-system independent format
2347          */
2348         outcount = 0;
2349         while (outcount < bytes_wanted) {
2350                 ino64_t objnum;
2351                 ushort_t reclen;
2352                 off64_t *next = NULL;

2354                 /*
2355                  * Special case '.', '..', and '.zfs'.
2356                  */
2357                 if (offset == 0) {
2358                         (void) strcpy(zap.za_name, ".");
2359                         zap.za_normalization_conflict = 0;
2360                         objnum = zp->z_id;
2361                 } else if (offset == 1) {
2362                         (void) strcpy(zap.za_name, "..");
2363                         zap.za_normalization_conflict = 0;
2364                         objnum = parent;
2365                 } else if (offset == 2 && zfs_show_ctldir(zp)) {
2366                         (void) strcpy(zap.za_name, ZFS_CTLDIR_NAME);
2367                         zap.za_normalization_conflict = 0;
2368                         objnum = ZFSCTL_INO_ROOT;
2369                 } else {
2370                         /*
2371                          * Grab next entry.
```

```
2372                         */
2373                        if (error = zap_cursor_retrieve(&zc, &zap)) {
2374                                if ((*eofp = (error == ENOENT)) != 0)
2375                                        break;
2376                                else
2377                                        goto update;
2378                        }
2379
2380                        if (zap.za_integer_length != 8 ||
2381                            zap.za_num_integers != 1) {
2382                                cmn_err(CE_WARN, "zap_readdir: bad directory "
2383                                    "entry, obj = %lld, offset = %lld\n",
2384                                    (u_longlong_t)zp->z_id,
2385                                    (u_longlong_t)offset);
2386                                error = SET_ERROR(ENXIO);
2387                                goto update;
2388                        }
2389
2390                        objnum = ZFS_DIRENT_OBJ(zap.za_first_integer);
2391                        /*
2392                         * MacOS X can extract the object type here such as:
2393                         * uint8_t type = ZFS_DIRENT_TYPE(zap.za_first_integer);
2394                         */
2395
2396                        if (check_sysattrs && !zap.za_normalization_conflict) {
2397                                zap.za_normalization_conflict =
2398                                    xattr_sysattr_casechk(zap.za_name);
2399                        }
2400                }
2401
2402                if (flags & V_RDDIR_ACCFILTER) {
2403                        /*
2404                         * If we have no access at all, don't include
2405                         * this entry in the returned information
2406                         */
2407                        znode_t *ezp;
2408                        if (zfs_zget(zp->z_zfsvfs, objnum, &ezp) != 0)
2409                                goto skip_entry;
2410                        if (!zfs_has_access(ezp, cr)) {
2411                                VN_RELE(ZTOV(ezp));
2412                                goto skip_entry;
2413                        }
2414                        VN_RELE(ZTOV(ezp));
2415                }
2416
2417                if (flags & V_RDDIR_ENTFLAGS)
2418                        reclen = EDIRENT_RECLEN(strlen(zap.za_name));
2419                else
2420                        reclen = DIRENT64_RECLEN(strlen(zap.za_name));
2421
2422                /*
2423                 * Will this entry fit in the buffer?
2424                 */
2425                if (outcount + reclen > bufsize) {
2426                        /*
2427                         * Did we manage to fit anything in the buffer?
2428                         */
2429                        if (!outcount) {
2430                                error = SET_ERROR(EINVAL);
2431                                goto update;
2432                        }
2433                        break;
2434                }
2435                if (flags & V_RDDIR_ENTFLAGS) {
2436                        /*
2437                         * Add extended flag entry:
```



```
2438                         */
2439                        eodp->ed_ino = objnum;
2440                        eodp->ed_reclen = reclen;
2441                        /* NOTE: ed_off is the offset for the *next* entry */
2442                        next = &(eodp->ed_off);
2443                        eodp->ed_eflags = zap.za_normalization_conflict ?
2444                            ED_CASE_CONFLICT : 0;
2445                        (void) strncpy(eodp->ed_name, zap.za_name,
2446                            EDIRENT_NAMELEN(reclen));
2447                        eodp = (edirent_t *)((intptr_t)eodp + reclen);
2448                } else {
2449                        /*
2450                         * Add normal entry:
2451                         */
2452                        odp->d_ino = objnum;
2453                        odp->d_reclen = reclen;
2454                        /* NOTE: d_off is the offset for the *next* entry */
2455                        next = &(odp->d_off);
2456                        (void) strncpy(odp->d_name, zap.za_name,
2457                            DIRENT64_NAMELEN(reclen));
2458                        odp = (dirent64_t *)((intptr_t)odp + reclen);
2459                }
2460                outcount += reclen;
2461
2462                ASSERT(outcount <= bufsize);
2463
2464                /* Prefetch znode */
2465                if (prefetch)
2466                        dmu_prefetch(os, objnum, 0, 0);
2467
2468        skip_entry:
2469                /*
2470                 * Move to the next entry, fill in the previous offset.
2471                 */
2472                if (offset > 2 || (offset == 2 && !zfs_show_ctldir(zp))) {
2473                        zap_cursor_advance(&zc);
2474                        offset = zap_cursor_serialize(&zc);
2475                } else {
2476                        offset += 1;
2477                }
2478                if (next)
2479                        *next = offset;
2480        }
2481        zp->z_zn_prefetch = B_FALSE; /* a lookup will re-enable pre-fetching */
2482
2483        if (uio->uio_segflg == UIO_SYSSPACE && uio->uio_iovcnt == 1) {
2484                iovp->iov_base += outcount;
2485                iovp->iov_len -= outcount;
2486                uio->uio_resid -= outcount;
2487        } else if (error = uiomove(outbuf, (long)outcount, UIO_READ, uio)) {
2488                /*
2489                 * Reset the pointer.
2490                 */
2491                offset = uio->uio_loffset;
2492        }
2493
2494 update:
2495        zap_cursor_fini(&zc);
2496        if (uio->uio_segflg != UIO_SYSSPACE || uio->uio_iovcnt != 1)
2497                kmem_free(outbuf, bufsize);
2498
2499        if (error == ENOENT)
2500                error = 0;
2501
2502        ZFS_ACCESSTIME_STAMP(zfsvfs, zp);
```

```
2504         uio->uio_loffset = offset;
2505         ZFS_EXIT(zfsvfs);
2506         return (error);
2507 }

2509 ulong_t zfs_fsync_sync_cnt = 4;

2511 static int
2512 zfs_fsync(vnode_t *vp, int syncflag, cred_t *cr, caller_context_t *ct)
2513 {
2514         znode_t *zp = VTOZ(vp);
2515         zfsvfs_t *zfsvfs = zp->z_zfsvfs;

2517         /*
2518          * Regardless of whether this is required for standards conformance,
2519          * this is the logical behavior when fsync() is called on a file with
2520          * dirty pages.  We use B_ASYNC since the ZIL transactions are already
2521          * going to be pushed out as part of the zil_commit().
2522          */
2523         if (vn_has_cached_data(vp) && !(syncflag & FNODSYNC) &&
2524             (vp->v_type == VREG) && !(IS_SWAPVP(vp)))
2525                 (void) VOP_PUTPAGE(vp, (offset_t)0, (size_t)0, B_ASYNC, cr, ct);

2527         (void) tsd_set(zfs_fsyncer_key, (void *)zfs_fsync_sync_cnt);

2529         if (zfsvfs->z_os->os_sync != ZFS_SYNC_DISABLED) {
2530                 ZFS_ENTER(zfsvfs);
2531                 ZFS_VERIFY_ZP(zp);
2532                 zil_commit(zfsvfs->z_log, zp->z_id);
2533                 ZFS_EXIT(zfsvfs);
2534         }
2535         return (0);
2536 }


2539 /*
2540  * Get the requested file attributes and place them in the provided
2541  * vattr structure.
2542  *
2543  *      IN:     vp      - vnode of file.
2544  *              vap     - va_mask identifies requested attributes.
2545  *                        If AT_XVATTR set, then optional attrs are requested
2546  *              flags   - ATTR_NOACLCHECK (CIFS server context)
2547  *              cr      - credentials of caller.
2548  *              ct      - caller context
2549  *
2550  *      OUT:    vap     - attribute values.
2551  *
2552  *      RETURN: 0 (always succeeds).
2553  */
2554 /* ARGSUSED */
2555 static int
2556 zfs_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2557     caller_context_t *ct)
2558 {
2559         znode_t *zp = VTOZ(vp);
2560         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
2561         int     error = 0;
2562         uint64_t links;
2563         uint64_t mtime[2], ctime[2];
2564         xvattr_t *xvap = (xvattr_t *)vap;       /* vap may be an xvattr_t * */
2565         xoptattr_t *xoap = NULL;
2566         boolean_t skipaclchk = (flags & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
2567         sa_bulk_attr_t bulk[2];
2568         int count = 0;
```



```
2570         ZFS_ENTER(zfsvfs);
2571         ZFS_VERIFY_ZP(zp);

2573         zfs_fuid_map_ids(zp, cr, &vap->va_uid, &vap->va_gid);

2575         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL, &mtime, 16);
2576         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL, &ctime, 16);

2578         if ((error = sa_bulk_lookup(zp->z_sa_hdl, bulk, count)) != 0) {
2579                 ZFS_EXIT(zfsvfs);
2580                 return (error);
2581         }

2583         /*
2584          * If ACL is trivial don't bother looking for ACE_READ_ATTRIBUTES.
2585          * Also, if we are the owner don't bother, since owner should
2586          * always be allowed to read basic attributes of file.
2587          */
2588         if (!(zp->z_pflags & ZFS_ACL_TRIVIAL) &&
2589             (vap->va_uid != crgetuid(cr))) {
2590                 if (error = zfs_zaccess(zp, ACE_READ_ATTRIBUTES, 0,
2591                     skipaclchk, cr)) {
2592                         ZFS_EXIT(zfsvfs);
2593                         return (error);
2594                 }
2595         }

2597         /*
2598          * Return all attributes.  It's cheaper to provide the answer
2599          * than to determine whether we were asked the question.
2600          */

2602         mutex_enter(&zp->z_lock);
2603         vap->va_type = vp->v_type;
2604         vap->va_mode = zp->z_mode & MODEMASK;
2605         vap->va_fsid = zp->z_zfsvfs->z_vfs->vfs_dev;
2606         vap->va_nodeid = zp->z_id;
2607         if ((vp->v_flag & VROOT) && zfs_show_ctldir(zp))
2608                 links = zp->z_links + 1;
2609         else
2610                 links = zp->z_links;
2611         vap->va_nlink = MIN(links, UINT32_MAX); /* nlink_t limit! */
2612         vap->va_size = zp->z_size;
2613         vap->va_rdev = vp->v_rdev;
2614         vap->va_seq = zp->z_seq;

2616         /*
2617          * Add in any requested optional attributes and the create time.
2618          * Also set the corresponding bits in the returned attribute bitmap.
2619          */
2620         if ((xoap = xva_getxoptattr(xvap)) != NULL && zfsvfs->z_use_fuids) {
2621                 if (XVA_ISSET_REQ(xvap, XAT_ARCHIVE)) {
2622                         xoap->xoa_archive =
2623                             ((zp->z_pflags & ZFS_ARCHIVE) != 0);
2624                         XVA_SET_RTN(xvap, XAT_ARCHIVE);
2625                 }

2627                 if (XVA_ISSET_REQ(xvap, XAT_READONLY)) {
2628                         xoap->xoa_readonly =
2629                             ((zp->z_pflags & ZFS_READONLY) != 0);
2630                         XVA_SET_RTN(xvap, XAT_READONLY);
2631                 }

2633                 if (XVA_ISSET_REQ(xvap, XAT_SYSTEM)) {
2634                         xoap->xoa_system =
2635                             ((zp->z_pflags & ZFS_SYSTEM) != 0);
```

```
2636                    XVA_SET_RTN(xvap, XAT_SYSTEM);
2637            }
2639            if (XVA_ISSET_REQ(xvap, XAT_HIDDEN)) {
2640                    xoap->xoa_hidden =
2641                        ((zp->z_pflags & ZFS_HIDDEN) != 0);
2642                    XVA_SET_RTN(xvap, XAT_HIDDEN);
2643            }
2645            if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2646                    xoap->xoa_nounlink =
2647                        ((zp->z_pflags & ZFS_NOUNLINK) != 0);
2648                    XVA_SET_RTN(xvap, XAT_NOUNLINK);
2649            }
2651            if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2652                    xoap->xoa_immutable =
2653                        ((zp->z_pflags & ZFS_IMMUTABLE) != 0);
2654                    XVA_SET_RTN(xvap, XAT_IMMUTABLE);
2655            }
2657            if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2658                    xoap->xoa_appendonly =
2659                        ((zp->z_pflags & ZFS_APPENDONLY) != 0);
2660                    XVA_SET_RTN(xvap, XAT_APPENDONLY);
2661            }
2663            if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2664                    xoap->xoa_nodump =
2665                        ((zp->z_pflags & ZFS_NODUMP) != 0);
2666                    XVA_SET_RTN(xvap, XAT_NODUMP);
2667            }
2669            if (XVA_ISSET_REQ(xvap, XAT_OPAQUE)) {
2670                    xoap->xoa_opaque =
2671                        ((zp->z_pflags & ZFS_OPAQUE) != 0);
2672                    XVA_SET_RTN(xvap, XAT_OPAQUE);
2673            }
2675            if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
2676                    xoap->xoa_av_quarantined =
2677                        ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0);
2678                    XVA_SET_RTN(xvap, XAT_AV_QUARANTINED);
2679            }
2681            if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
2682                    xoap->xoa_av_modified =
2683                        ((zp->z_pflags & ZFS_AV_MODIFIED) != 0);
2684                    XVA_SET_RTN(xvap, XAT_AV_MODIFIED);
2685            }
2687            if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) &&
2688                vp->v_type == VREG) {
2689                    zfs_sa_get_scanstamp(zp, xvap);
2690            }
2692            if (XVA_ISSET_REQ(xvap, XAT_CREATETIME)) {
2693                    uint64_t times[2];
2695                    (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_CRTIME(zfsvfs),
2696                        times, sizeof (times));
2697                    ZFS_TIME_DECODE(&xoap->xoa_createtime, times);
2698                    XVA_SET_RTN(xvap, XAT_CREATETIME);
2699            }
2701            if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
```



```
2702                    xoap->xoa_reparse = ((zp->z_pflags & ZFS_REPARSE) != 0);
2703                    XVA_SET_RTN(xvap, XAT_REPARSE);
2704            }
2705            if (XVA_ISSET_REQ(xvap, XAT_GEN)) {
2706                    xoap->xoa_generation = zp->z_gen;
2707                    XVA_SET_RTN(xvap, XAT_GEN);
2708            }
2710            if (XVA_ISSET_REQ(xvap, XAT_OFFLINE)) {
2711                    xoap->xoa_offline =
2712                        ((zp->z_pflags & ZFS_OFFLINE) != 0);
2713                    XVA_SET_RTN(xvap, XAT_OFFLINE);
2714            }
2716            if (XVA_ISSET_REQ(xvap, XAT_SPARSE)) {
2717                    xoap->xoa_sparse =
2718                        ((zp->z_pflags & ZFS_SPARSE) != 0);
2719                    XVA_SET_RTN(xvap, XAT_SPARSE);
2720            }
2721    }
2723    ZFS_TIME_DECODE(&vap->va_atime, zp->z_atime);
2724    ZFS_TIME_DECODE(&vap->va_mtime, mtime);
2725    ZFS_TIME_DECODE(&vap->va_ctime, ctime);
2727    mutex_exit(&zp->z_lock);
2729    sa_object_size(zp->z_sa_hdl, &vap->va_blksize, &vap->va_nblocks);
2731    if (zp->z_blksz == 0) {
2732            /*
2733             * Block size hasn't been set; suggest maximal I/O transfers.
2734             */
2735            vap->va_blksize = zfsvfs->z_max_blksz;
2736    }
2738    ZFS_EXIT(zfsvfs);
2739    return (0);
2740 }
2742 /*
2743  * Set the file attributes to the values contained in the
2744  * vattr structure.
2745  *
2746  *      IN:     vp      - vnode of file to be modified.
2747  *              vap     - new attribute values.
2748  *                        If AT_XVATTR set, then optional attrs are being set
2749  *              flags   - ATTR_UTIME set if non-default time values provided.
2750  *                      - ATTR_NOACLCHECK (CIFS context only).
2751  *              cr      - credentials of caller.
2752  *              ct      - caller context
2753  *
2754  *      RETURN: 0 on success, error code on failure.
2755  *
2756  * Timestamps:
2757  *      vp - ctime updated, mtime updated if size changed.
2758  */
2759 /* ARGSUSED */
2760 static int
2761 zfs_setattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2762     caller_context_t *ct)
2763 {
2764    znode_t         *zp = VTOZ(vp);
2765    zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2766    zilog_t         *zilog;
2767    dmu_tx_t        *tx;
```

```
2768        vattr_t         oldva;
2769        xvattr_t        tmpxvattr;
2770        uint_t          mask = vap->va_mask;
2771        uint_t          saved_mask = 0;
2772        int             trim_mask = 0;
2773        uint64_t        new_mode;
2774        uint64_t        new_uid, new_gid;
2775        uint64_t        xattr_obj;
2776        uint64_t        mtime[2], ctime[2];
2777        znode_t         *attrzp;
2778        int             need_policy = FALSE;
2779        int             err, err2;
2780        zfs_fuid_info_t *fuidp = NULL;
2781        xvattr_t *xvap = (xvattr_t *)vap;       /* vap may be an xvattr_t * */
2782        xoptattr_t      *xoap;
2783        zfs_acl_t       *aclp;
2784        boolean_t skipaclchk = (flags & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
2785        boolean_t       fuid_dirtied = B_FALSE;
2786        sa_bulk_attr_t  bulk[7], xattr_bulk[7];
2787        int             count = 0, xattr_count = 0;

2789        if (mask == 0)
2790                return (0);

2792        if (mask & AT_NOSET)
2793                return (SET_ERROR(EINVAL));

2795        ZFS_ENTER(zfsvfs);
2796        ZFS_VERIFY_ZP(zp);

2798        zilog = zfsvfs->z_log;

2800        /*
2801         * Make sure that if we have ephemeral uid/gid or xvattr specified
2802         * that file system is at proper version level
2803         */

2805        if (zfsvfs->z_use_fuids == B_FALSE &&
2806            (((mask & AT_UID) && IS_EPHEMERAL(vap->va_uid)) ||
2807            ((mask & AT_GID) && IS_EPHEMERAL(vap->va_gid)) ||
2808            (mask & AT_XVATTR))) {
2809                ZFS_EXIT(zfsvfs);
2810                return (SET_ERROR(EINVAL));
2811        }

2813        if (mask & AT_SIZE && vp->v_type == VDIR) {
2814                ZFS_EXIT(zfsvfs);
2815                return (SET_ERROR(EISDIR));
2816        }

2818        if (mask & AT_SIZE && vp->v_type != VREG && vp->v_type != VFIFO) {
2819                ZFS_EXIT(zfsvfs);
2820                return (SET_ERROR(EINVAL));
2821        }

2823        /*
2824         * If this is an xvattr_t, then get a pointer to the structure of
2825         * optional attributes.  If this is NULL, then we have a vattr_t.
2826         */
2827        xoap = xva_getxoptattr(xvap);

2829        xva_init(&tmpxvattr);

2831        /*
2832         * Immutable files can only alter immutable bit and atime
2833         */
```



```
2834        if ((zp->z_pflags & ZFS_IMMUTABLE) &&
2835            ((mask & (AT_SIZE|AT_UID|AT_GID|AT_MTIME|AT_MODE)) ||
2836            ((mask & AT_XVATTR) && XVA_ISSET_REQ(xvap, XAT_CREATETIME)))) {
2837                ZFS_EXIT(zfsvfs);
2838                return (SET_ERROR(EPERM));
2839        }

2841        if ((mask & AT_SIZE) && (zp->z_pflags & ZFS_READONLY)) {
2842                ZFS_EXIT(zfsvfs);
2843                return (SET_ERROR(EPERM));
2844        }

2846        /*
2847         * Verify timestamps doesn't overflow 32 bits.
2848         * ZFS can handle large timestamps, but 32bit syscalls can't
2849         * handle times greater than 2039.  This check should be removed
2850         * once large timestamps are fully supported.
2851         */
2852        if (mask & (AT_ATIME | AT_MTIME)) {
2853                if (((mask & AT_ATIME) && TIMESPEC_OVERFLOW(&vap->va_atime)) ||
2854                    ((mask & AT_MTIME) && TIMESPEC_OVERFLOW(&vap->va_mtime))) {
2855                        ZFS_EXIT(zfsvfs);
2856                        return (SET_ERROR(EOVERFLOW));
2857                }
2858        }

2860 top:
2861        attrzp = NULL;
2862        aclp = NULL;

2864        /* Can this be moved to before the top label? */
2865        if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
2866                ZFS_EXIT(zfsvfs);
2867                return (SET_ERROR(EROFS));
2868        }

2870        /*
2871         * First validate permissions
2872         */

2874        if (mask & AT_SIZE) {
2875                err = zfs_zaccess(zp, ACE_WRITE_DATA, 0, skipaclchk, cr);
2876                if (err) {
2877                        ZFS_EXIT(zfsvfs);
2878                        return (err);
2879                }
2880                /*
2881                 * XXX - Note, we are not providing any open
2882                 * mode flags here (like FNDELAY), so we may
2883                 * block if there are locks present... this
2884                 * should be addressed in openat().
2885                 */
2886                /* XXX - would it be OK to generate a log record here? */
2887                err = zfs_freesp(zp, vap->va_size, 0, 0, FALSE);
2888                if (err) {
2889                        ZFS_EXIT(zfsvfs);
2890                        return (err);
2891                }

2893                if (vap->va_size == 0)
2894                        vnevent_truncate(ZTOV(zp), ct);
2895        }

2897        if (mask & (AT_ATIME|AT_MTIME) ||
2898            ((mask & AT_XVATTR) && (XVA_ISSET_REQ(xvap, XAT_HIDDEN) ||
2899            XVA_ISSET_REQ(xvap, XAT_READONLY) ||
```

```
2900             XVA_ISSET_REQ(xvap, XAT_ARCHIVE) ||
2901             XVA_ISSET_REQ(xvap, XAT_OFFLINE) ||
2902             XVA_ISSET_REQ(xvap, XAT_SPARSE) ||
2903             XVA_ISSET_REQ(xvap, XAT_CREATETIME) ||
2904             XVA_ISSET_REQ(xvap, XAT_SYSTEM))) {
2905                 need_policy = zfs_zaccess(zp, ACE_WRITE_ATTRIBUTES, 0,
2906                     skipaclchk, cr);
2907         }
2908
2909         if (mask & (AT_UID|AT_GID)) {
2910                 int     idmask = (mask & (AT_UID|AT_GID));
2911                 int     take_owner;
2912                 int     take_group;
2913
2914                 /*
2915                  * NOTE: even if a new mode is being set,
2916                  * we may clear S_ISUID/S_ISGID bits.
2917                  */
2918
2919                 if (!(mask & AT_MODE))
2920                         vap->va_mode = zp->z_mode;
2921
2922                 /*
2923                  * Take ownership or chgrp to group we are a member of
2924                  */
2925
2926                 take_owner = (mask & AT_UID) && (vap->va_uid == crgetuid(cr));
2927                 take_group = (mask & AT_GID) &&
2928                     zfs_groupmember(zfsvfs, vap->va_gid, cr);
2929
2930                 /*
2931                  * If both AT_UID and AT_GID are set then take_owner and
2932                  * take_group must both be set in order to allow taking
2933                  * ownership.
2934                  *
2935                  * Otherwise, send the check through secpolicy_vnode_setattr()
2936                  *
2937                  */
2938
2939                 if (((idmask == (AT_UID|AT_GID)) && take_owner && take_group) ||
2940                     ((idmask == AT_UID) && take_owner) ||
2941                     ((idmask == AT_GID) && take_group)) {
2942                         if (zfs_zaccess(zp, ACE_WRITE_OWNER, 0,
2943                             skipaclchk, cr) == 0) {
2944                                 /*
2945                                  * Remove setuid/setgid for non-privileged users
2946                                  */
2947                                 secpolicy_setid_clear(vap, cr);
2948                                 trim_mask = (mask & (AT_UID|AT_GID));
2949                         } else {
2950                                 need_policy =  TRUE;
2951                         }
2952                 } else {
2953                         need_policy =  TRUE;
2954                 }
2955         }
2956
2957         mutex_enter(&zp->z_lock);
2958         oldva.va_mode = zp->z_mode;
2959         zfs_fuid_map_ids(zp, cr, &oldva.va_uid, &oldva.va_gid);
2960         if (mask & AT_XVATTR) {
2961                 /*
2962                  * Update xvattr mask to include only those attributes
2963                  * that are actually changing.
2964                  *
2965                  * the bits will be restored prior to actually setting
```



```
2966                  * the attributes so the caller thinks they were set.
2967                  */
2968                 if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2969                         if (xoap->xoa_appendonly !=
2970                             ((zp->z_pflags & ZFS_APPENDONLY) != 0)) {
2971                                 need_policy = TRUE;
2972                         } else {
2973                                 XVA_CLR_REQ(xvap, XAT_APPENDONLY);
2974                                 XVA_SET_REQ(&tmpxvattr, XAT_APPENDONLY);
2975                         }
2976                 }
2977
2978                 if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2979                         if (xoap->xoa_nounlink !=
2980                             ((zp->z_pflags & ZFS_NOUNLINK) != 0)) {
2981                                 need_policy = TRUE;
2982                         } else {
2983                                 XVA_CLR_REQ(xvap, XAT_NOUNLINK);
2984                                 XVA_SET_REQ(&tmpxvattr, XAT_NOUNLINK);
2985                         }
2986                 }
2987
2988                 if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2989                         if (xoap->xoa_immutable !=
2990                             ((zp->z_pflags & ZFS_IMMUTABLE) != 0)) {
2991                                 need_policy = TRUE;
2992                         } else {
2993                                 XVA_CLR_REQ(xvap, XAT_IMMUTABLE);
2994                                 XVA_SET_REQ(&tmpxvattr, XAT_IMMUTABLE);
2995                         }
2996                 }
2997
2998                 if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2999                         if (xoap->xoa_nodump !=
3000                             ((zp->z_pflags & ZFS_NODUMP) != 0)) {
3001                                 need_policy = TRUE;
3002                         } else {
3003                                 XVA_CLR_REQ(xvap, XAT_NODUMP);
3004                                 XVA_SET_REQ(&tmpxvattr, XAT_NODUMP);
3005                         }
3006                 }
3007
3008                 if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
3009                         if (xoap->xoa_av_modified !=
3010                             ((zp->z_pflags & ZFS_AV_MODIFIED) != 0)) {
3011                                 need_policy = TRUE;
3012                         } else {
3013                                 XVA_CLR_REQ(xvap, XAT_AV_MODIFIED);
3014                                 XVA_SET_REQ(&tmpxvattr, XAT_AV_MODIFIED);
3015                         }
3016                 }
3017
3018                 if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
3019                         if ((vp->v_type != VREG &&
3020                             xoap->xoa_av_quarantined) ||
3021                             xoap->xoa_av_quarantined !=
3022                             ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0)) {
3023                                 need_policy = TRUE;
3024                         } else {
3025                                 XVA_CLR_REQ(xvap, XAT_AV_QUARANTINED);
3026                                 XVA_SET_REQ(&tmpxvattr, XAT_AV_QUARANTINED);
3027                         }
3028                 }
3029
3030                 if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
3031                         mutex_exit(&zp->z_lock);
```

```
3032                        ZFS_EXIT(zfsvfs);
3033                        return (SET_ERROR(EPERM));
3034                }

3036                if (need_policy == FALSE &&
3037                    (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) ||
3038                    XVA_ISSET_REQ(xvap, XAT_OPAQUE))) {
3039                        need_policy = TRUE;
3040                }
3041        }

3043        mutex_exit(&zp->z_lock);

3045        if (mask & AT_MODE) {
3046                if (zfs_zaccess(zp, ACE_WRITE_ACL, 0, skipaclchk, cr) == 0) {
3047                        err = secpolicy_setid_setsticky_clear(vp, vap,
3048                            &oldva, cr);
3049                        if (err) {
3050                                ZFS_EXIT(zfsvfs);
3051                                return (err);
3052                        }
3053                        trim_mask |= AT_MODE;
3054                } else {
3055                        need_policy = TRUE;
3056                }
3057        }

3059        if (need_policy) {
3060                /*
3061                 * If trim_mask is set then take ownership
3062                 * has been granted or write_acl is present and user
3063                 * has the ability to modify mode.  In that case remove
3064                 * UID|GID and or MODE from mask so that
3065                 * secpolicy_vnode_setattr() doesn't revoke it.
3066                 */

3068                if (trim_mask) {
3069                        saved_mask = vap->va_mask;
3070                        vap->va_mask &= ~trim_mask;
3071                }
3072                err = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
3073                    (int (*)(void *, int, cred_t *))zfs_zaccess_unix, zp);
3074                if (err) {
3075                        ZFS_EXIT(zfsvfs);
3076                        return (err);
3077                }

3079                if (trim_mask)
3080                        vap->va_mask |= saved_mask;
3081        }

3083        /*
3084         * secpolicy_vnode_setattr, or take ownership may have
3085         * changed va_mask
3086         */
3087        mask = vap->va_mask;

3089        if ((mask & (AT_UID | AT_GID))) {
3090                err = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
3091                    &xattr_obj, sizeof (xattr_obj));

3093                if (err == 0 && xattr_obj) {
3094                        err = zfs_zget(zp->z_zfsvfs, xattr_obj, &attrzp);
3095                        if (err)
3096                                goto out2;
3097                }
```



```
3098                if (mask & AT_UID) {
3099                        new_uid = zfs_fuid_create(zfsvfs,
3100                            (uint64_t)vap->va_uid, cr, ZFS_OWNER, &fuidp);
3101                        if (new_uid != zp->z_uid &&
3102                            zfs_fuid_overquota(zfsvfs, B_FALSE, new_uid)) {
3103                                if (attrzp)
3104                                        VN_RELE(ZTOV(attrzp));
3105                                err = SET_ERROR(EDQUOT);
3106                                goto out2;
3107                        }
3108                }

3110                if (mask & AT_GID) {
3111                        new_gid = zfs_fuid_create(zfsvfs, (uint64_t)vap->va_gid,
3112                            cr, ZFS_GROUP, &fuidp);
3113                        if (new_gid != zp->z_gid &&
3114                            zfs_fuid_overquota(zfsvfs, B_TRUE, new_gid)) {
3115                                if (attrzp)
3116                                        VN_RELE(ZTOV(attrzp));
3117                                err = SET_ERROR(EDQUOT);
3118                                goto out2;
3119                        }
3120                }
3121        }
3122        tx = dmu_tx_create(zfsvfs->z_os);

3124        if (mask & AT_MODE) {
3125                uint64_t pmode = zp->z_mode;
3126                uint64_t acl_obj;
3127                new_mode = (pmode & S_IFMT) | (vap->va_mode & ~S_IFMT);

3129                if (zp->z_zfsvfs->z_acl_mode == ZFS_ACL_RESTRICTED &&
3130                    !(zp->z_pflags & ZFS_ACL_TRIVIAL)) {
3131                        err = SET_ERROR(EPERM);
3132                        goto out;
3133                }

3135                if (err = zfs_acl_chmod_setattr(zp, &aclp, new_mode))
3136                        goto out;

3138                mutex_enter(&zp->z_lock);
3139                if (!zp->z_is_sa && ((acl_obj = zfs_external_acl(zp)) != 0)) {
3140                        /*
3141                         * Are we upgrading ACL from old V0 format
3142                         * to V1 format?
3143                         */
3144                        if (zfsvfs->z_version >= ZPL_VERSION_FUID &&
3145                            zfs_znode_acl_version(zp) ==
3146                            ZFS_ACL_VERSION_INITIAL) {
3147                                dmu_tx_hold_free(tx, acl_obj, 0,
3148                                    DMU_OBJECT_END);
3149                                dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3150                                    0, aclp->z_acl_bytes);
3151                        } else {
3152                                dmu_tx_hold_write(tx, acl_obj, 0,
3153                                    aclp->z_acl_bytes);
3154                        }
3155                } else if (!zp->z_is_sa && aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3156                        dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3157                            0, aclp->z_acl_bytes);
3158                }
3159                mutex_exit(&zp->z_lock);
3160                dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3161        } else {
3162                if ((mask & AT_XVATTR) &&
3163                    XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
```

```
3164                        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3165                else
3166                        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
3167        }

3169        if (attrzp) {
3170                dmu_tx_hold_sa(tx, attrzp->z_sa_hdl, B_FALSE);
3171        }

3173        fuid_dirtied = zfsvfs->z_fuid_dirty;
3174        if (fuid_dirtied)
3175                zfs_fuid_txhold(zfsvfs, tx);

3177        zfs_sa_upgrade_txholds(tx, zp);

3179        err = dmu_tx_assign(tx, TXG_WAIT);
3180        if (err)
3181                goto out;

3183        count = 0;
3184        /*
3185         * Set each attribute requested.
3186         * We group settings according to the locks they need to acquire.
3187         *
3188         * Note: you cannot set ctime directly, although it will be
3189         * updated as a side-effect of calling this function.
3190         */


3193        if (mask & (AT_UID|AT_GID|AT_MODE))
3194                mutex_enter(&zp->z_acl_lock);
3195        mutex_enter(&zp->z_lock);

3197        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
3198            &zp->z_pflags, sizeof (zp->z_pflags));

3200        if (attrzp) {
3201                if (mask & (AT_UID|AT_GID|AT_MODE))
3202                        mutex_enter(&attrzp->z_acl_lock);
3203                mutex_enter(&attrzp->z_lock);
3204                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3205                    SA_ZPL_FLAGS(zfsvfs), NULL, &attrzp->z_pflags,
3206                    sizeof (attrzp->z_pflags));
3207        }

3209        if (mask & (AT_UID|AT_GID)) {

3211                if (mask & AT_UID) {
3212                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_UID(zfsvfs), NULL,
3213                            &new_uid, sizeof (new_uid));
3214                        zp->z_uid = new_uid;
3215                        if (attrzp) {
3216                                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3217                                    SA_ZPL_UID(zfsvfs), NULL, &new_uid,
3218                                    sizeof (new_uid));
3219                                attrzp->z_uid = new_uid;
3220                        }
3221                }

3223                if (mask & AT_GID) {
3224                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_GID(zfsvfs),
3225                            NULL, &new_gid, sizeof (new_gid));
3226                        zp->z_gid = new_gid;
3227                        if (attrzp) {
3228                                SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3229                                    SA_ZPL_GID(zfsvfs), NULL, &new_gid,
```



```
3230                                    sizeof (new_gid));
3231                                attrzp->z_gid = new_gid;
3232                        }
3233                }
3234                if (!(mask & AT_MODE)) {
3235                        SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs),
3236                            NULL, &new_mode, sizeof (new_mode));
3237                        new_mode = zp->z_mode;
3238                }
3239                err = zfs_acl_chown_setattr(zp);
3240                ASSERT(err == 0);
3241                if (attrzp) {
3242                        err = zfs_acl_chown_setattr(attrzp);
3243                        ASSERT(err == 0);
3244                }
3245        }

3247        if (mask & AT_MODE) {
3248                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs), NULL,
3249                    &new_mode, sizeof (new_mode));
3250                zp->z_mode = new_mode;
3251                ASSERT3U((uintptr_t)aclp, !=, NULL);
3252                err = zfs_aclset_common(zp, aclp, cr, tx);
3253                ASSERT0(err);
3254                if (zp->z_acl_cached)
3255                        zfs_acl_free(zp->z_acl_cached);
3256                zp->z_acl_cached = aclp;
3257                aclp = NULL;
3258        }


3261        if (mask & AT_ATIME) {
3262                ZFS_TIME_ENCODE(&vap->va_atime, zp->z_atime);
3263                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ATIME(zfsvfs), NULL,
3264                    &zp->z_atime, sizeof (zp->z_atime));
3265        }

3267        if (mask & AT_MTIME) {
3268                ZFS_TIME_ENCODE(&vap->va_mtime, mtime);
3269                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
3270                    mtime, sizeof (mtime));
3271        }

3273        /* XXX - shouldn't this be done *before* the ATIME/MTIME checks? */
3274        if (mask & AT_SIZE && !(mask & AT_MTIME)) {
3275                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs),
3276                    NULL, mtime, sizeof (mtime));
3277                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3278                    &ctime, sizeof (ctime));
3279                zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
3280                    B_TRUE);
3281        } else if (mask != 0) {
3282                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
3283                    &ctime, sizeof (ctime));
3284                zfs_tstamp_update_setup(zp, STATE_CHANGED, mtime, ctime,
3285                    B_TRUE);
3286                if (attrzp) {
3287                        SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3288                            SA_ZPL_CTIME(zfsvfs), NULL,
3289                            &ctime, sizeof (ctime));
3290                        zfs_tstamp_update_setup(attrzp, STATE_CHANGED,
3291                            mtime, ctime, B_TRUE);
3292                }
3293        }
3294        /*
3295         * Do this after setting timestamps to prevent timestamp
```

```
3296          * update from toggling bit
3297          */
3299         if (xoap && (mask & AT_XVATTR)) {
3301                 /*
3302                  * restore trimmed off masks
3303                  * so that return masks can be set for caller.
3304                  */
3306                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_APPENDONLY)) {
3307                         XVA_SET_REQ(xvap, XAT_APPENDONLY);
3308                 }
3309                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_NOUNLINK)) {
3310                         XVA_SET_REQ(xvap, XAT_NOUNLINK);
3311                 }
3312                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_IMMUTABLE)) {
3313                         XVA_SET_REQ(xvap, XAT_IMMUTABLE);
3314                 }
3315                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_NODUMP)) {
3316                         XVA_SET_REQ(xvap, XAT_NODUMP);
3317                 }
3318                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_MODIFIED)) {
3319                         XVA_SET_REQ(xvap, XAT_AV_MODIFIED);
3320                 }
3321                 if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_QUARANTINED)) {
3322                         XVA_SET_REQ(xvap, XAT_AV_QUARANTINED);
3323                 }
3325                 if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3326                         ASSERT(vp->v_type == VREG);
3328                 zfs_xvattr_set(zp, xvap, tx);
3329         }
3331         if (fuid_dirtied)
3332                 zfs_fuid_sync(zfsvfs, tx);
3334         if (mask != 0)
3335                 zfs_log_setattr(zilog, tx, TX_SETATTR, zp, vap, mask, fuidp);
3337         mutex_exit(&zp->z_lock);
3338         if (mask & (AT_UID|AT_GID|AT_MODE))
3339                 mutex_exit(&zp->z_acl_lock);
3341         if (attrzp) {
3342                 if (mask & (AT_UID|AT_GID|AT_MODE))
3343                         mutex_exit(&attrzp->z_acl_lock);
3344                 mutex_exit(&attrzp->z_lock);
3345         }
3346 out:
3347         if (err == 0 && attrzp) {
3348                 err2 = sa_bulk_update(attrzp->z_sa_hdl, xattr_bulk,
3349                     xattr_count, tx);
3350                 ASSERT(err2 == 0);
3351         }
3353         if (attrzp)
3354                 VN_RELE(ZTOV(attrzp));
3356         if (aclp)
3357                 zfs_acl_free(aclp);
3359         if (fuidp) {
3360                 zfs_fuid_info_free(fuidp);
3361                 fuidp = NULL;
```



```
3362         }
3364         if (err) {
3365                 dmu_tx_abort(tx);
3366                 if (err == ERESTART)
3367                         goto top;
3368         } else {
3369                 err2 = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
3370                 dmu_tx_commit(tx);
3371         }
3373 out2:
3374         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3375                 zil_commit(zilog, 0);
3377         ZFS_EXIT(zfsvfs);
3378         return (err);
3379 }
3381 typedef struct zfs_zlock {
3382         krwlock_t       *zl_rwlock;     /* lock we acquired */
3383         znode_t         *zl_znode;      /* znode we held */
3384         struct zfs_zlock *zl_next;      /* next in list */
3385 } zfs_zlock_t;
3387 /*
3388  * Drop locks and release vnodes that were held by zfs_rename_lock().
3389  */
3390 static void
3391 zfs_rename_unlock(zfs_zlock_t **zlpp)
3392 {
3393         zfs_zlock_t *zl;
3395         while ((zl = *zlpp) != NULL) {
3396                 if (zl->zl_znode != NULL)
3397                         VN_RELE(ZTOV(zl->zl_znode));
3398                 rw_exit(zl->zl_rwlock);
3399                 *zlpp = zl->zl_next;
3400                 kmem_free(zl, sizeof (*zl));
3401         }
3402 }
3404 /*
3405  * Search back through the directory tree, using the ".." entries.
3406  * Lock each directory in the chain to prevent concurrent renames.
3407  * Fail any attempt to move a directory into one of its own descendants.
3408  * XXX - z_parent_lock can overlap with map or grow locks
3409  */
3410 static int
3411 zfs_rename_lock(znode_t *szp, znode_t *tdzp, znode_t *sdzp, zfs_zlock_t **zlpp)
3412 {
3413         zfs_zlock_t     *zl;
3414         znode_t         *zp = tdzp;
3415         uint64_t        rootid = zp->z_zfsvfs->z_root;
3416         uint64_t        oidp = zp->z_id;
3417         krwlock_t       *rwlp = &szp->z_parent_lock;
3418         krw_t           rw = RW_WRITER;
3420         /*
3421          * First pass write-locks szp and compares to zp->z_id.
3422          * Later passes read-lock zp and compare to zp->z_parent.
3423          */
3424         do {
3425                 if (!rw_tryenter(rwlp, rw)) {
3426                         /*
3427                          * Another thread is renaming in this path.
```

```
3428                         * Note that if we are a WRITER, we don't have any
3429                         * parent_locks held yet.
3430                         */
3431                        if (rw == RW_READER && zp->z_id > szp->z_id) {
3432                                /*
3433                                 * Drop our locks and restart
3434                                 */
3435                                zfs_rename_unlock(&zl);
3436                                *zlpp = NULL;
3437                                zp = tdzp;
3438                                oidp = zp->z_id;
3439                                rwlp = &szp->z_parent_lock;
3440                                rw = RW_WRITER;
3441                                continue;
3442                        } else {
3443                                /*
3444                                 * Wait for other thread to drop its locks
3445                                 */
3446                                rw_enter(rwlp, rw);
3447                        }
3448                }

3450                zl = kmem_alloc(sizeof (*zl), KM_SLEEP);
3451                zl->zl_rwlock = rwlp;
3452                zl->zl_znode = NULL;
3453                zl->zl_next = *zlpp;
3454                *zlpp = zl;

3456                if (oidp == szp->z_id)          /* We're a descendant of szp */
3457                        return (SET_ERROR(EINVAL));

3459                if (oidp == rootid)             /* We've hit the top */
3460                        return (0);

3462                if (rw == RW_READER) {          /* i.e. not the first pass */
3463                        int error = zfs_zget(zp->z_zfsvfs, oidp, &zp);
3464                        if (error)
3465                                return (error);
3466                        zl->zl_znode = zp;
3467                }
3468                (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_PARENT(zp->z_zfsvfs),
3469                    &oidp, sizeof (oidp));
3470                rwlp = &zp->z_parent_lock;
3471                rw = RW_READER;

3473        } while (zp->z_id != sdzp->z_id);

3475        return (0);
3476 }

3478 /*
3479  * Move an entry from the provided source directory to the target
3480  * directory.  Change the entry name as indicated.
3481  *
3482  *      IN:     sdvp    - Source directory containing the "old entry".
3483  *              snm     - Old entry name.
3484  *              tdvp    - Target directory to contain the "new entry".
3485  *              tnm     - New entry name.
3486  *              cr      - credentials of caller.
3487  *              ct      - caller context
3488  *              flags   - case flags
3489  *
3490  *      RETURN: 0 on success, error code on failure.
3491  *
3492  * Timestamps:
3493  *      sdvp,tdvp - ctime|mtime updated
```



```
3494  */
3495 /*ARGSUSED*/
3496 static int
3497 zfs_rename(vnode_t *sdvp, char *snm, vnode_t *tdvp, char *tnm, cred_t *cr,
3498     caller_context_t *ct, int flags)
3499 {
3500         znode_t         *tdzp, *szp, *tzp;
3501         znode_t         *sdzp = VTOZ(sdvp);
3502         zfsvfs_t        *zfsvfs = sdzp->z_zfsvfs;
3503         zilog_t         *zilog;
3504         vnode_t         *realvp;
3505         zfs_dirlock_t   *sdl, *tdl;
3506         dmu_tx_t        *tx;
3507         zfs_zlock_t     *zl;
3508         int             cmp, serr, terr;
3509         int             error = 0;
3510         int             zflg = 0;
3511         boolean_t       waited = B_FALSE;

3513         ZFS_ENTER(zfsvfs);
3514         ZFS_VERIFY_ZP(sdzp);
3515         zilog = zfsvfs->z_log;

3517         /*
3518          * Make sure we have the real vp for the target directory.
3519          */
3520         if (VOP_REALVP(tdvp, &realvp, ct) == 0)
3521                 tdvp = realvp;

3523         tdzp = VTOZ(tdvp);
3524         ZFS_VERIFY_ZP(tdzp);

3526         /*
3527          * We check z_zfsvfs rather than v_vfsp here, because snapshots and the
3528          * ctldir appear to have the same v_vfsp.
3529          */
3530         if (tdzp->z_zfsvfs != zfsvfs || zfsctl_is_node(tdvp)) {
3531                 ZFS_EXIT(zfsvfs);
3532                 return (SET_ERROR(EXDEV));
3533         }

3535         if (zfsvfs->z_utf8 && u8_validate(tnm,
3536             strlen(tnm), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3537                 ZFS_EXIT(zfsvfs);
3538                 return (SET_ERROR(EILSEQ));
3539         }

3541         if (flags & FIGNORECASE)
3542                 zflg |= ZCILOOK;

3544 top:
3545         szp = NULL;
3546         tzp = NULL;
3547         zl = NULL;

3549         /*
3550          * This is to prevent the creation of links into attribute space
3551          * by renaming a linked file into/outof an attribute directory.
3552          * See the comment in zfs_link() for why this is considered bad.
3553          */
3554         if ((tdzp->z_pflags & ZFS_XATTR) != (sdzp->z_pflags & ZFS_XATTR)) {
3555                 ZFS_EXIT(zfsvfs);
3556                 return (SET_ERROR(EINVAL));
3557         }

3559         /*
```

```
3560          * Lock source and target directory entries.  To prevent deadlock,
3561          * a lock ordering must be defined.  We lock the directory with
3562          * the smallest object id first, or if it's a tie, the one with
3563          * the lexically first name.
3564          */
3565         if (sdzp->z_id < tdzp->z_id) {
3566                 cmp = -1;
3567         } else if (sdzp->z_id > tdzp->z_id) {
3568                 cmp = 1;
3569         } else {
3570                 /*
3571                  * First compare the two name arguments without
3572                  * considering any case folding.
3573                  */
3574                 int nofold = (zfsvfs->z_norm & ~U8_TEXTPREP_TOUPPER);

3576                 cmp = u8_strcmp(snm, tnm, 0, nofold, U8_UNICODE_LATEST, &error);
3577                 ASSERT(error == 0 || !zfsvfs->z_utf8);
3578                 if (cmp == 0) {
3579                         /*
3580                          * POSIX: "If the old argument and the new argument
3581                          * both refer to links to the same existing file,
3582                          * the rename() function shall return successfully
3583                          * and perform no other action."
3584                          */
3585                         ZFS_EXIT(zfsvfs);
3586                         return (0);
3587                 }
3588                 /*
3589                  * If the file system is case-folding, then we may
3590                  * have some more checking to do.  A case-folding file
3591                  * system is either supporting mixed case sensitivity
3592                  * access or is completely case-insensitive.  Note
3593                  * that the file system is always case preserving.
3594                  *
3595                  * In mixed sensitivity mode case sensitive behavior
3596                  * is the default.  FIGNORECASE must be used to
3597                  * explicitly request case insensitive behavior.
3598                  *
3599                  * If the source and target names provided differ only
3600                  * by case (e.g., a request to rename 'tim' to 'Tim'),
3601                  * we will treat this as a special case in the
3602                  * case-insensitive mode: as long as the source name
3603                  * is an exact match, we will allow this to proceed as
3604                  * a name-change request.
3605                  */
3606                 if ((zfsvfs->z_case == ZFS_CASE_INSENSITIVE ||
3607                     (zfsvfs->z_case == ZFS_CASE_MIXED &&
3608                     flags & FIGNORECASE)) &&
3609                     u8_strcmp(snm, tnm, 0, zfsvfs->z_norm, U8_UNICODE_LATEST,
3610                     &error) == 0) {
3611                         /*
3612                          * case preserving rename request, require exact
3613                          * name matches
3614                          */
3615                         zflg |= ZCIEXACT;
3616                         zflg &= ~ZCILOOK;
3617                 }
3618         }

3620         /*
3621          * If the source and destination directories are the same, we should
3622          * grab the z_name_lock of that directory only once.
3623          */
3624         if (sdzp == tdzp) {
3625                 zflg |= ZHAVELOCK;
```



```
3626                 rw_enter(&sdzp->z_name_lock, RW_READER);
3627         }

3629         if (cmp < 0) {
3630                 serr = zfs_dirent_lock(&sdl, sdzp, snm, &szp,
3631                     ZEXISTS | zflg, NULL, NULL);
3632                 terr = zfs_dirent_lock(&tdl,
3633                     tdzp, tnm, &tzp, ZRENAMING | zflg, NULL, NULL);
3634         } else {
3635                 terr = zfs_dirent_lock(&tdl,
3636                     tdzp, tnm, &tzp, zflg, NULL, NULL);
3637                 serr = zfs_dirent_lock(&sdl,
3638                     sdzp, snm, &szp, ZEXISTS | ZRENAMING | zflg,
3639                     NULL, NULL);
3640         }

3642         if (serr) {
3643                 /*
3644                  * Source entry invalid or not there.
3645                  */
3646                 if (!terr) {
3647                         zfs_dirent_unlock(tdl);
3648                         if (tzp)
3649                                 VN_RELE(ZTOV(tzp));
3650                 }

3652                 if (sdzp == tdzp)
3653                         rw_exit(&sdzp->z_name_lock);

3655                 if (strcmp(snm, "..") == 0)
3656                         serr = SET_ERROR(EINVAL);
3657                 ZFS_EXIT(zfsvfs);
3658                 return (serr);
3659         }
3660         if (terr) {
3661                 zfs_dirent_unlock(sdl);
3662                 VN_RELE(ZTOV(szp));

3664                 if (sdzp == tdzp)
3665                         rw_exit(&sdzp->z_name_lock);

3667                 if (strcmp(tnm, "..") == 0)
3668                         terr = SET_ERROR(EINVAL);
3669                 ZFS_EXIT(zfsvfs);
3670                 return (terr);
3671         }

3673         /*
3674          * Must have write access at the source to remove the old entry
3675          * and write access at the target to create the new entry.
3676          * Note that if target and source are the same, this can be
3677          * done in a single check.
3678          */

3680         if (error = zfs_zaccess_rename(sdzp, szp, tdzp, tzp, cr))
3681                 goto out;

3683         if (ZTOV(szp)->v_type == VDIR) {
3684                 /*
3685                  * Check to make sure rename is valid.
3686                  * Can't do a move like this: /usr/a/b to /usr/a/b/c/d
3687                  */
3688                 if (error = zfs_rename_lock(szp, tdzp, sdzp, &zl))
3689                         goto out;
3690         }
```

```
3692         /*
3693          * Does target exist?
3694          */
3695         if (tzp) {
3696                 /*
3697                  * Source and target must be the same type.
3698                  */
3699                 if (ZTOV(szp)->v_type == VDIR) {
3700                         if (ZTOV(tzp)->v_type != VDIR) {
3701                                 error = SET_ERROR(ENOTDIR);
3702                                 goto out;
3703                         }
3704                 } else {
3705                         if (ZTOV(tzp)->v_type == VDIR) {
3706                                 error = SET_ERROR(EISDIR);
3707                                 goto out;
3708                         }
3709                 }
3710                 /*
3711                  * POSIX dictates that when the source and target
3712                  * entries refer to the same file object, rename
3713                  * must do nothing and exit without error.
3714                  */
3715                 if (szp->z_id == tzp->z_id) {
3716                         error = 0;
3717                         goto out;
3718                 }
3719         }

3721         vnevent_rename_src(ZTOV(szp), sdvp, snm, ct);
3722         if (tzp)
3723                 vnevent_rename_dest(ZTOV(tzp), tdvp, tnm, ct);

3725         /*
3726          * notify the target directory if it is not the same
3727          * as source directory.
3728          */
3729         if (tdvp != sdvp) {
3730                 vnevent_rename_dest_dir(tdvp, ct);
3731         }

3733         tx = dmu_tx_create(zfsvfs->z_os);
3734         dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
3735         dmu_tx_hold_sa(tx, sdzp->z_sa_hdl, B_FALSE);
3736         dmu_tx_hold_zap(tx, sdzp->z_id, FALSE, snm);
3737         dmu_tx_hold_zap(tx, tdzp->z_id, TRUE, tnm);
3738         if (sdzp != tdzp) {
3739                 dmu_tx_hold_sa(tx, tdzp->z_sa_hdl, B_FALSE);
3740                 zfs_sa_upgrade_txholds(tx, tdzp);
3741         }
3742         if (tzp) {
3743                 dmu_tx_hold_sa(tx, tzp->z_sa_hdl, B_FALSE);
3744                 zfs_sa_upgrade_txholds(tx, tzp);
3745         }

3747         zfs_sa_upgrade_txholds(tx, szp);
3748         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
3749         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
3750         if (error) {
3751                 if (zl != NULL)
3752                         zfs_rename_unlock(&zl);
3753                 zfs_dirent_unlock(sdl);
3754                 zfs_dirent_unlock(tdl);

3756                 if (sdzp == tdzp)
3757                         rw_exit(&sdzp->z_name_lock);
```



```
3759                 VN_RELE(ZTOV(szp));
3760                 if (tzp)
3761                         VN_RELE(ZTOV(tzp));
3762                 if (error == ERESTART) {
3763                         waited = B_TRUE;
3764                         dmu_tx_wait(tx);
3765                         dmu_tx_abort(tx);
3766                         goto top;
3767                 }
3768                 dmu_tx_abort(tx);
3769                 ZFS_EXIT(zfsvfs);
3770                 return (error);
3771         }

3773         if (tzp)        /* Attempt to remove the existing target */
3774                 error = zfs_link_destroy(tdl, tzp, tx, zflg, NULL);

3776         if (error == 0) {
3777                 error = zfs_link_create(tdl, szp, tx, ZRENAMING);
3778                 if (error == 0) {
3779                         szp->z_pflags |= ZFS_AV_MODIFIED;

3781                         error = sa_update(szp->z_sa_hdl, SA_ZPL_FLAGS(zfsvfs),
3782                             (void *)&szp->z_pflags, sizeof (uint64_t), tx);
3783                         ASSERT0(error);

3785                         error = zfs_link_destroy(sdl, szp, tx, ZRENAMING, NULL);
3786                         if (error == 0) {
3787                                 zfs_log_rename(zilog, tx, TX_RENAME |
3788                                     (flags & FIGNORECASE ? TX_CI : 0), sdzp,
3789                                     sdl->dl_name, tdzp, tdl->dl_name, szp);

3791                                 /*
3792                                  * Update path information for the target vnode
3793                                  */
3794                                 vn_renamepath(tdvp, ZTOV(szp), tnm,
3795                                     strlen(tnm));
3796                         } else {
3797                                 /*
3798                                  * At this point, we have successfully created
3799                                  * the target name, but have failed to remove
3800                                  * the source name.  Since the create was done
3801                                  * with the ZRENAMING flag, there are
3802                                  * complications; for one, the link count is
3803                                  * wrong.  The easiest way to deal with this
3804                                  * is to remove the newly created target, and
3805                                  * return the original error.  This must
3806                                  * succeed; fortunately, it is very unlikely to
3807                                  * fail, since we just created it.
3808                                  */
3809                                 VERIFY3U(zfs_link_destroy(tdl, szp, tx,
3810                                     ZRENAMING, NULL), ==, 0);
3811                         }
3812                 }
3813         }

3815         dmu_tx_commit(tx);
3816 out:
3817         if (zl != NULL)
3818                 zfs_rename_unlock(&zl);

3820         zfs_dirent_unlock(sdl);
3821         zfs_dirent_unlock(tdl);

3823         if (sdzp == tdzp)
```

```
3824                 rw_exit(&sdzp->z_name_lock);


3827         VN_RELE(ZTOV(szp));
3828         if (tzp)
3829                 VN_RELE(ZTOV(tzp));

3831         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3832                 zil_commit(zilog, 0);

3834         ZFS_EXIT(zfsvfs);
3835         return (error);
3836 }

3838 /*
3839  * Insert the indicated symbolic reference entry into the directory.
3840  *
3841  *      IN:     dvp     - Directory to contain new symbolic link.
3842  *              link    - Name for new symlink entry.
3843  *              vap     - Attributes of new entry.
3844  *              cr      - credentials of caller.
3845  *              ct      - caller context
3846  *              flags   - case flags
3847  *
3848  *      RETURN: 0 on success, error code on failure.
3849  *
3850  * Timestamps:
3851  *      dvp - ctime|mtime updated
3852  */
3853 /*ARGSUSED*/
3854 static int
3855 zfs_symlink(vnode_t *dvp, char *name, vattr_t *vap, char *link, cred_t *cr,
3856     caller_context_t *ct, int flags)
3857 {
3858         znode_t         *zp, *dzp = VTOZ(dvp);
3859         zfs_dirlock_t   *dl;
3860         dmu_tx_t        *tx;
3861         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
3862         zilog_t         *zilog;
3863         uint64_t        len = strlen(link);
3864         int             error;
3865         int             zflg = ZNEW;
3866         zfs_acl_ids_t   acl_ids;
3867         boolean_t       fuid_dirtied;
3868         uint64_t        txtype = TX_SYMLINK;
3869         boolean_t       waited = B_FALSE;

3871         ASSERT(vap->va_type == VLNK);

3873         ZFS_ENTER(zfsvfs);
3874         ZFS_VERIFY_ZP(dzp);
3875         zilog = zfsvfs->z_log;

3877         if (zfsvfs->z_utf8 && u8_validate(name, strlen(name),
3878             NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3879                 ZFS_EXIT(zfsvfs);
3880                 return (SET_ERROR(EILSEQ));
3881         }
3882         if (flags & FIGNORECASE)
3883                 zflg |= ZCILOOK;

3885         if (len > MAXPATHLEN) {
3886                 ZFS_EXIT(zfsvfs);
3887                 return (SET_ERROR(ENAMETOOLONG));
3888         }
```



```
3890         if ((error = zfs_acl_ids_create(dzp, 0,
3891             vap, cr, NULL, &acl_ids)) != 0) {
3892                 ZFS_EXIT(zfsvfs);
3893                 return (error);
3894         }
3895 top:
3896         /*
3897          * Attempt to lock directory; fail if entry already exists.
3898          */
3899         error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg, NULL, NULL);
3900         if (error) {
3901                 zfs_acl_ids_free(&acl_ids);
3902                 ZFS_EXIT(zfsvfs);
3903                 return (error);
3904         }

3906         if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
3907                 zfs_acl_ids_free(&acl_ids);
3908                 zfs_dirent_unlock(dl);
3909                 ZFS_EXIT(zfsvfs);
3910                 return (error);
3911         }

3913         if (zfs_acl_ids_overquota(zfsvfs, &acl_ids)) {
3914                 zfs_acl_ids_free(&acl_ids);
3915                 zfs_dirent_unlock(dl);
3916                 ZFS_EXIT(zfsvfs);
3917                 return (SET_ERROR(EDQUOT));
3918         }
3919         tx = dmu_tx_create(zfsvfs->z_os);
3920         fuid_dirtied = zfsvfs->z_fuid_dirty;
3921         dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0, MAX(1, len));
3922         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
3923         dmu_tx_hold_sa_create(tx, acl_ids.z_aclp->z_acl_bytes +
3924             ZFS_SA_BASE_ATTR_SIZE + len);
3925         dmu_tx_hold_sa(tx, dzp->z_sa_hdl, B_FALSE);
3926         if (!zfsvfs->z_use_sa && acl_ids.z_aclp->z_acl_bytes > ZFS_ACE_SPACE) {
3927                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT, 0,
3928                     acl_ids.z_aclp->z_acl_bytes);
3929         }
3930         if (fuid_dirtied)
3931                 zfs_fuid_txhold(zfsvfs, tx);
3932         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
3933         if (error) {
3934                 zfs_dirent_unlock(dl);
3935                 if (error == ERESTART) {
3936                         waited = B_TRUE;
3937                         dmu_tx_wait(tx);
3938                         dmu_tx_abort(tx);
3939                         goto top;
3940                 }
3941                 zfs_acl_ids_free(&acl_ids);
3942                 dmu_tx_abort(tx);
3943                 ZFS_EXIT(zfsvfs);
3944                 return (error);
3945         }

3947         /*
3948          * Create a new object for the symlink.
3949          * for version 4 ZPL datsets the symlink will be an SA attribute
3950          */
3951         zfs_mknode(dzp, vap, tx, cr, 0, &zp, &acl_ids);

3953         if (fuid_dirtied)
3954                 zfs_fuid_sync(zfsvfs, tx);
```

```
3956         mutex_enter(&zp->z_lock);
3957         if (zp->z_is_sa)
3958                 error = sa_update(zp->z_sa_hdl, SA_ZPL_SYMLINK(zfsvfs),
3959                     link, len, tx);
3960         else
3961                 zfs_sa_symlink(zp, link, len, tx);
3962         mutex_exit(&zp->z_lock);

3964         zp->z_size = len;
3965         (void) sa_update(zp->z_sa_hdl, SA_ZPL_SIZE(zfsvfs),
3966             &zp->z_size, sizeof (zp->z_size), tx);
3967         /*
3968          * Insert the new object into the directory.
3969          */
3970         (void) zfs_link_create(dl, zp, tx, ZNEW);

3972         if (flags & FIGNORECASE)
3973                 txtype |= TX_CI;
3974         zfs_log_symlink(zilog, tx, txtype, dzp, zp, name, link);

3976         zfs_acl_ids_free(&acl_ids);

3978         dmu_tx_commit(tx);

3980         zfs_dirent_unlock(dl);

3982         VN_RELE(ZTOV(zp));

3984         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3985                 zil_commit(zilog, 0);

3987         ZFS_EXIT(zfsvfs);
3988         return (error);
3989 }

3991 /*
3992  * Return, in the buffer contained in the provided uio structure,
3993  * the symbolic path referred to by vp.
3994  *
3995  *      IN:     vp      - vnode of symbolic link.
3996  *              uio     - structure to contain the link path.
3997  *              cr      - credentials of caller.
3998  *              ct      - caller context
3999  *
4000  *      OUT:    uio     - structure containing the link path.
4001  *
4002  *      RETURN: 0 on success, error code on failure.
4003  *
4004  * Timestamps:
4005  *      vp - atime updated
4006  */
4007 /* ARGSUSED */
4008 static int
4009 zfs_readlink(vnode_t *vp, uio_t *uio, cred_t *cr, caller_context_t *ct)
4010 {
4011         znode_t         *zp = VTOZ(vp);
4012         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4013         int             error;

4015         ZFS_ENTER(zfsvfs);
4016         ZFS_VERIFY_ZP(zp);

4018         mutex_enter(&zp->z_lock);
4019         if (zp->z_is_sa)
4020                 error = sa_lookup_uio(zp->z_sa_hdl,
4021                     SA_ZPL_SYMLINK(zfsvfs), uio);
```



```
4022         else
4023                 error = zfs_sa_readlink(zp, uio);
4024         mutex_exit(&zp->z_lock);

4026         ZFS_ACCESSTIME_STAMP(zfsvfs, zp);

4028         ZFS_EXIT(zfsvfs);
4029         return (error);
4030 }

4032 /*
4033  * Insert a new entry into directory tdvp referencing svp.
4034  *
4035  *      IN:     tdvp    - Directory to contain new entry.
4036  *              svp     - vnode of new entry.
4037  *              name    - name of new entry.
4038  *              cr      - credentials of caller.
4039  *              ct      - caller context
4040  *
4041  *      RETURN: 0 on success, error code on failure.
4042  *
4043  * Timestamps:
4044  *      tdvp - ctime|mtime updated
4045  *       svp - ctime updated
4046  */
4047 /* ARGSUSED */
4048 static int
4049 zfs_link(vnode_t *tdvp, vnode_t *svp, char *name, cred_t *cr,
4050     caller_context_t *ct, int flags)
4051 {
4052         znode_t         *dzp = VTOZ(tdvp);
4053         znode_t         *tzp, *szp;
4054         zfsvfs_t        *zfsvfs = dzp->z_zfsvfs;
4055         zilog_t         *zilog;
4056         zfs_dirlock_t   *dl;
4057         dmu_tx_t        *tx;
4058         vnode_t         *realvp;
4059         int             error;
4060         int             zf = ZNEW;
4061         uint64_t        parent;
4062         uid_t           owner;
4063         boolean_t       waited = B_FALSE;

4065         ASSERT(tdvp->v_type == VDIR);

4067         ZFS_ENTER(zfsvfs);
4068         ZFS_VERIFY_ZP(dzp);
4069         zilog = zfsvfs->z_log;

4071         if (VOP_REALVP(svp, &realvp, ct) == 0)
4072                 svp = realvp;

4074         /*
4075          * POSIX dictates that we return EPERM here.
4076          * Better choices include ENOTSUP or EISDIR.
4077          */
4078         if (svp->v_type == VDIR) {
4079                 ZFS_EXIT(zfsvfs);
4080                 return (SET_ERROR(EPERM));
4081         }

4083         szp = VTOZ(svp);
4084         ZFS_VERIFY_ZP(szp);

4086         /*
4087          * We check z_zfsvfs rather than v_vfsp here, because snapshots and the
```

```
4088          * ctldir appear to have the same v_vfsp.
4089          */
4090         if (szp->z_zfsvfs != zfsvfs || zfsctl_is_node(svp)) {
4091                 ZFS_EXIT(zfsvfs);
4092                 return (SET_ERROR(EXDEV));
4093         }

4095         /* Prevent links to .zfs/shares files */

4097         if ((error = sa_lookup(szp->z_sa_hdl, SA_ZPL_PARENT(zfsvfs),
4098             &parent, sizeof (uint64_t))) != 0) {
4099                 ZFS_EXIT(zfsvfs);
4100                 return (error);
4101         }
4102         if (parent == zfsvfs->z_shares_dir) {
4103                 ZFS_EXIT(zfsvfs);
4104                 return (SET_ERROR(EPERM));
4105         }

4107         if (zfsvfs->z_utf8 && u8_validate(name,
4108             strlen(name), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
4109                 ZFS_EXIT(zfsvfs);
4110                 return (SET_ERROR(EILSEQ));
4111         }
4112         if (flags & FIGNORECASE)
4113                 zf |= ZCILOOK;

4115         /*
4116          * We do not support links between attributes and non-attributes
4117          * because of the potential security risk of creating links
4118          * into "normal" file space in order to circumvent restrictions
4119          * imposed in attribute space.
4120          */
4121         if ((szp->z_pflags & ZFS_XATTR) != (dzp->z_pflags & ZFS_XATTR)) {
4122                 ZFS_EXIT(zfsvfs);
4123                 return (SET_ERROR(EINVAL));
4124         }


4127         owner = zfs_fuid_map_id(zfsvfs, szp->z_uid, cr, ZFS_OWNER);
4128         if (owner != crgetuid(cr) && secpolicy_basic_link(cr) != 0) {
4129                 ZFS_EXIT(zfsvfs);
4130                 return (SET_ERROR(EPERM));
4131         }

4133         if (error = zfs_zaccess(dzp, ACE_ADD_FILE, 0, B_FALSE, cr)) {
4134                 ZFS_EXIT(zfsvfs);
4135                 return (error);
4136         }

4138 top:
4139         /*
4140          * Attempt to lock directory; fail if entry already exists.
4141          */
4142         error = zfs_dirent_lock(&dl, dzp, name, &tzp, zf, NULL, NULL);
4143         if (error) {
4144                 ZFS_EXIT(zfsvfs);
4145                 return (error);
4146         }

4148         tx = dmu_tx_create(zfsvfs->z_os);
4149         dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
4150         dmu_tx_hold_zap(tx, dzp->z_id, TRUE, name);
4151         zfs_sa_upgrade_txholds(tx, szp);
4152         zfs_sa_upgrade_txholds(tx, dzp);
4153         error = dmu_tx_assign(tx, waited ? TXG_WAITED : TXG_NOWAIT);
```



```
4154         if (error) {
4155                 zfs_dirent_unlock(dl);
4156                 if (error == ERESTART) {
4157                         waited = B_TRUE;
4158                         dmu_tx_wait(tx);
4159                         dmu_tx_abort(tx);
4160                         goto top;
4161                 }
4162                 dmu_tx_abort(tx);
4163                 ZFS_EXIT(zfsvfs);
4164                 return (error);
4165         }

4167         error = zfs_link_create(dl, szp, tx, 0);

4169         if (error == 0) {
4170                 uint64_t txtype = TX_LINK;
4171                 if (flags & FIGNORECASE)
4172                         txtype |= TX_CI;
4173                 zfs_log_link(zilog, tx, txtype, dzp, szp, name);
4174         }

4176         dmu_tx_commit(tx);

4178         zfs_dirent_unlock(dl);

4180         if (error == 0) {
4181                 vnevent_link(svp, ct);
4182         }

4184         if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
4185                 zil_commit(zilog, 0);

4187         ZFS_EXIT(zfsvfs);
4188         return (error);
4189 }

4191 /*
4192  * zfs_null_putapage() is used when the file system has been force
4193  * unmounted. It just drops the pages.
4194  */
4195 /* ARGSUSED */
4196 static int
4197 zfs_null_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp,
4198                 size_t *lenp, int flags, cred_t *cr)
4199 {
4200         pvn_write_done(pp, B_INVAL|B_FORCE|B_ERROR);
4201         return (0);
4202 }

4204 /*
4205  * Push a page out to disk, klustering if possible.
4206  *
4207  *      IN:     vp      - file to push page to.
4208  *              pp      - page to push.
4209  *              flags   - additional flags.
4210  *              cr      - credentials of caller.
4211  *
4212  *      OUT:    offp    - start of range pushed.
4213  *              lenp    - len of range pushed.
4214  *
4215  *      RETURN: 0 on success, error code on failure.
4216  *
4217  * NOTE: callers must have locked the page to be pushed.  On
4218  * exit, the page (and all other pages in the kluster) must be
4219  * unlocked.
```

```
 */
/* ARGSUSED */
static int
zfs_putapage(vnode_t *vp, page_t *pp, u_offset_t *offp,
                size_t *lenp, int flags, cred_t *cr)
{
        znode_t         *zp = VTOZ(vp);
        zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
        dmu_tx_t        *tx;
        u_offset_t      off, koff;
        size_t          len, klen;
        int             err;

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

        /*
         * Can't push pages past end-of-file.
         */
        if (off >= zp->z_size) {
                /* ignore all pages */
                err = 0;
                goto out;
        } else if (off + len > zp->z_size) {
                int npages = btopr(zp->z_size - off);
                page_t *trunc;

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

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

        dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
        zfs_sa_upgrade_txholds(tx, zp);
        err = dmu_tx_assign(tx, TXG_WAIT);
        if (err != 0) {
                dmu_tx_abort(tx);
                goto out;
        }

        if (zp->z_blksz <= PAGESIZE) {
                caddr_t va = zfs_map_page(pp, S_READ);
```



```
                ASSERT3U(len, <=, PAGESIZE);
                dmu_write(zfsvfs->z_os, zp->z_id, off, len, va, tx);
                zfs_unmap_page(pp, va);
        } else {
                err = dmu_write_pages(zfsvfs->z_os, zp->z_id, off, len, pp, tx);
        }

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

                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
                    &mtime, 16);
                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_CTIME(zfsvfs), NULL,
                    &ctime, 16);
                SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
                    &zp->z_pflags, 8);
                zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
                    B_TRUE);
                zfs_log_write(zfsvfs->z_log, tx, TX_WRITE, zp, off, len, 0);
        }
        dmu_tx_commit(tx);

out:
        pvn_write_done(pp, (err ? B_ERROR : 0) | flags);
        if (offp)
                *offp = off;
        if (lenp)
                *lenp = len;

        return (err);
}

/*
 * Copy the portion of the file indicated from pages into the file.
 * The pages are stored in a page list attached to the files vnode.
 *
 *      IN:     vp      - vnode of file to push page data to.
 *              off     - position in file to put data.
 *              len     - amount of data to write.
 *              flags   - flags to control the operation.
 *              cr      - credentials of caller.
 *              ct      - caller context.
 *
 *      RETURN: 0 on success, error code on failure.
 *
 * Timestamps:
 *      vp - ctime|mtime updated
 */
/*ARGSUSED*/
static int
zfs_putpage(vnode_t *vp, offset_t off, size_t len, int flags, cred_t *cr,
    caller_context_t *ct)
{
        znode_t         *zp = VTOZ(vp);
        zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
        page_t          *pp;
        size_t          io_len;
        u_offset_t      io_off;
        uint_t          blksz;
        rl_t            *rl;
        int             error = 0;

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

```
4353         /*
4354          * There's nothing to do if no data is cached.
4355          */
4356         if (!vn_has_cached_data(vp)) {
4357                 ZFS_EXIT(zfsvfs);
4358                 return (0);
4359         }
4360
4361         /*
4362          * Align this request to the file block size in case we kluster.
4363          * XXX - this can result in pretty aggresive locking, which can
4364          * impact simultanious read/write access.  One option might be
4365          * to break up long requests (len == 0) into block-by-block
4366          * operations to get narrower locking.
4367          */
4368         blksz = zp->z_blksz;
4369         if (ISP2(blksz))
4370                 io_off = P2ALIGN_TYPED(off, blksz, u_offset_t);
4371         else
4372                 io_off = 0;
4373         if (len > 0 && ISP2(blksz))
4374                 io_len = P2ROUNDUP_TYPED(len + (off - io_off), blksz, size_t);
4375         else
4376                 io_len = 0;
4377
4378         if (io_len == 0) {
4379                 /*
4380                  * Search the entire vp list for pages >= io_off.
4381                  */
4382                 rl = zfs_range_lock(zp, io_off, UINT64_MAX, RL_WRITER);
4383                 error = pvn_vplist_dirty(vp, io_off, zfs_putapage, flags, cr);
4384                 goto out;
4385         }
4386         rl = zfs_range_lock(zp, io_off, io_len, RL_WRITER);
4387
4388         if (off > zp->z_size) {
4389                 /* past end of file */
4390                 zfs_range_unlock(rl);
4391                 ZFS_EXIT(zfsvfs);
4392                 return (0);
4393         }
4394
4395         len = MIN(io_len, P2ROUNDUP(zp->z_size, PAGESIZE) - io_off);
4396
4397         for (off = io_off; io_off < off + len; io_off += io_len) {
4398                 if ((flags & B_INVAL) || ((flags & B_ASYNC) == 0)) {
4399                         pp = page_lookup(vp, io_off,
4400                             (flags & (B_INVAL | B_FREE)) ? SE_EXCL : SE_SHARED);
4401                 } else {
4402                         pp = page_lookup_nowait(vp, io_off,
4403                             (flags & B_FREE) ? SE_EXCL : SE_SHARED);
4404                 }
4405
4406                 if (pp != NULL && pvn_getdirty(pp, flags)) {
4407                         int err;
4408
4409                         /*
4410                          * Found a dirty page to push
4411                          */
4412                         err = zfs_putapage(vp, pp, &io_off, &io_len, flags, cr);
4413                         if (err)
4414                                 error = err;
4415                 } else {
4416                         io_len = PAGESIZE;
4417                 }
```



```
4418         }
4419 out:
4420         zfs_range_unlock(rl);
4421         if ((flags & B_ASYNC) == 0 || zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
4422                 zil_commit(zfsvfs->z_log, zp->z_id);
4423         ZFS_EXIT(zfsvfs);
4424         return (error);
4425 }
4426
4427 /*ARGSUSED*/
4428 void
4429 zfs_inactive(vnode_t *vp, cred_t *cr, caller_context_t *ct)
4430 {
4431         znode_t *zp = VTOZ(vp);
4432         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
4433         int error;
4434
4435         rw_enter(&zfsvfs->z_teardown_inactive_lock, RW_READER);
4436         if (zp->z_sa_hdl == NULL) {
4437                 /*
4438                  * The fs has been unmounted, or we did a
4439                  * suspend/resume and this file no longer exists.
4440                  */
4441                 if (vn_has_cached_data(vp)) {
4442                         (void) pvn_vplist_dirty(vp, 0, zfs_null_putapage,
4443                             B_INVAL, cr);
4444                 }
4445
4446                 mutex_enter(&zp->z_lock);
4447                 mutex_enter(&vp->v_lock);
4448                 ASSERT(vp->v_count == 1);
4449                 vp->v_count = 0;
4450                 mutex_exit(&vp->v_lock);
4451                 mutex_exit(&zp->z_lock);
4452                 rw_exit(&zfsvfs->z_teardown_inactive_lock);
4453                 zfs_znode_free(zp);
4454                 return;
4455         }
4456
4457         /*
4458          * Attempt to push any data in the page cache.  If this fails
4459          * we will get kicked out later in zfs_zinactive().
4460          */
4461         if (vn_has_cached_data(vp)) {
4462                 (void) pvn_vplist_dirty(vp, 0, zfs_putapage, B_INVAL|B_ASYNC,
4463                     cr);
4464         }
4465
4466         if (zp->z_atime_dirty && zp->z_unlinked == 0) {
4467                 dmu_tx_t *tx = dmu_tx_create(zfsvfs->z_os);
4468
4469                 dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
4470                 zfs_sa_upgrade_txholds(tx, zp);
4471                 error = dmu_tx_assign(tx, TXG_WAIT);
4472                 if (error) {
4473                         dmu_tx_abort(tx);
4474                 } else {
4475                         mutex_enter(&zp->z_lock);
4476                         (void) sa_update(zp->z_sa_hdl, SA_ZPL_ATIME(zfsvfs),
4477                             (void *)&zp->z_atime, sizeof (zp->z_atime), tx);
4478                         zp->z_atime_dirty = 0;
4479                         mutex_exit(&zp->z_lock);
4480                         dmu_tx_commit(tx);
4481                 }
4482         }
```

```
4484         zfs_zinactive(zp);
4485         rw_exit(&zfsvfs->z_teardown_inactive_lock);
4486 }

4488 /*
4489  * Bounds-check the seek operation.
4490  *
4491  *      IN:     vp      - vnode seeking within
4492  *              ooff    - old file offset
4493  *              noffp   - pointer to new file offset
4494  *              ct      - caller context
4495  *
4496  *      RETURN: 0 on success, EINVAL if new offset invalid.
4497  */
4498 /* ARGSUSED */
4499 static int
4500 zfs_seek(vnode_t *vp, offset_t ooff, offset_t *noffp,
4501     caller_context_t *ct)
4502 {
4503         if (vp->v_type == VDIR)
4504                 return (0);
4505         return ((*noffp < 0 || *noffp > MAXOFFSET_T) ? EINVAL : 0);
4506 }

4508 /*
4509  * Pre-filter the generic locking function to trap attempts to place
4510  * a mandatory lock on a memory mapped file.
4511  */
4512 static int
4513 zfs_frlock(vnode_t *vp, int cmd, flock64_t *bfp, int flag, offset_t offset,
4514     flk_callback_t *flk_cbp, cred_t *cr, caller_context_t *ct)
4515 {
4516         znode_t *zp = VTOZ(vp);
4517         zfsvfs_t *zfsvfs = zp->z_zfsvfs;

4519         ZFS_ENTER(zfsvfs);
4520         ZFS_VERIFY_ZP(zp);

4522         /*
4523          * We are following the UFS semantics with respect to mapcnt
4524          * here: If we see that the file is mapped already, then we will
4525          * return an error, but we don't worry about races between this
4526          * function and zfs_map().
4527          */
4528         if (zp->z_mapcnt > 0 && MANDMODE(zp->z_mode)) {
4529                 ZFS_EXIT(zfsvfs);
4530                 return (SET_ERROR(EAGAIN));
4531         }
4532         ZFS_EXIT(zfsvfs);
4533         return (fs_frlock(vp, cmd, bfp, flag, offset, flk_cbp, cr, ct));
4534 }

4536 /*
4537  * If we can't find a page in the cache, we will create a new page
4538  * and fill it with file data.  For efficiency, we may try to fill
4539  * multiple pages at once (klustering) to fill up the supplied page
4540  * list.  Note that the pages to be filled are held with an exclusive
4541  * lock to prevent access by other threads while they are being filled.
4542  */
4543 static int
4544 zfs_fillpage(vnode_t *vp, u_offset_t off, struct seg *seg,
4545     caddr_t addr, page_t *pl[], size_t plsz, enum seg_rw rw)
4546 {
4547         znode_t *zp = VTOZ(vp);
4548         page_t *pp, *cur_pp;
4549         objset_t *os = zp->z_zfsvfs->z_os;
```



```
4550         u_offset_t io_off, total;
4551         size_t io_len;
4552         int err;

4554         if (plsz == PAGESIZE || zp->z_blksz <= PAGESIZE) {
4555                 /*
4556                  * We only have a single page, don't bother klustering
4557                  */
4558                 io_off = off;
4559                 io_len = PAGESIZE;
4560                 pp = page_create_va(vp, io_off, io_len,
4561                     PG_EXCL | PG_WAIT, seg, addr);
4562         } else {
4563                 /*
4564                  * Try to find enough pages to fill the page list
4565                  */
4566                 pp = pvn_read_kluster(vp, off, seg, addr, &io_off,
4567                     &io_len, off, plsz, 0);
4568         }
4569         if (pp == NULL) {
4570                 /*
4571                  * The page already exists, nothing to do here.
4572                  */
4573                 *pl = NULL;
4574                 return (0);
4575         }

4577         /*
4578          * Fill the pages in the kluster.
4579          */
4580         cur_pp = pp;
4581         for (total = io_off + io_len; io_off < total; io_off += PAGESIZE) {
4582                 caddr_t va;

4584                 ASSERT3U(io_off, ==, cur_pp->p_offset);
4585                 va = zfs_map_page(cur_pp, S_WRITE);
4586                 err = dmu_read(os, zp->z_id, io_off, PAGESIZE, va,
4587                     DMU_READ_PREFETCH);
4588                 zfs_unmap_page(cur_pp, va);
4589                 if (err) {
4590                         /* On error, toss the entire kluster */
4591                         pvn_read_done(pp, B_ERROR);
4592                         /* convert checksum errors into IO errors */
4593                         if (err == ECKSUM)
4594                                 err = SET_ERROR(EIO);
4595                         return (err);
4596                 }
4597                 cur_pp = cur_pp->p_next;
4598         }

4600         /*
4601          * Fill in the page list array from the kluster starting
4602          * from the desired offset 'off'.
4603          * NOTE: the page list will always be null terminated.
4604          */
4605         pvn_plist_init(pp, pl, plsz, off, io_len, rw);
4606         ASSERT(pl == NULL || (*pl)->p_offset == off);

4608         return (0);
4609 }

4611 /*
4612  * Return pointers to the pages for the file region [off, off + len]
4613  * in the pl array.  If plsz is greater than len, this function may
4614  * also return page pointers from after the specified region
4615  * (i.e. the region [off, off + plsz]).  These additional pages are
```

```
4616  * only returned if they are already in the cache, or were created as
4617  * part of a klustered read.
4618  *
4619  *      IN:     vp      - vnode of file to get data from.
4620  *              off     - position in file to get data from.
4621  *              len     - amount of data to retrieve.
4622  *              plsz    - length of provided page list.
4623  *              seg     - segment to obtain pages for.
4624  *              addr    - virtual address of fault.
4625  *              rw      - mode of created pages.
4626  *              cr      - credentials of caller.
4627  *              ct      - caller context.
4628  *
4629  *      OUT:    protp   - protection mode of created pages.
4630  *              pl      - list of pages created.
4631  *
4632  *      RETURN: 0 on success, error code on failure.
4633  *
4634  * Timestamps:
4635  *      vp - atime updated
4636  */
4637 /* ARGSUSED */
4638 static int
4639 zfs_getpage(vnode_t *vp, offset_t off, size_t len, uint_t *protp,
4640     page_t *pl[], size_t plsz, struct seg *seg, caddr_t addr,
4641     enum seg_rw rw, cred_t *cr, caller_context_t *ct)
4642 {
4643         znode_t         *zp = VTOZ(vp);
4644         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4645         page_t          **pl0 = pl;
4646         int             err = 0;

4648         /* we do our own caching, faultahead is unnecessary */
4649         if (pl == NULL)
4650                 return (0);
4651         else if (len > plsz)
4652                 len = plsz;
4653         else
4654                 len = P2ROUNDUP(len, PAGESIZE);
4655         ASSERT(plsz >= len);

4657         ZFS_ENTER(zfsvfs);
4658         ZFS_VERIFY_ZP(zp);

4660         if (protp)
4661                 *protp = PROT_ALL;

4663         /*
4664          * Loop through the requested range [off, off + len) looking
4665          * for pages.  If we don't find a page, we will need to create
4666          * a new page and fill it with data from the file.
4667          */
4668         while (len > 0) {
4669                 if (*pl = page_lookup(vp, off, SE_SHARED))
4670                         *(pl+1) = NULL;
4671                 else if (err = zfs_fillpage(vp, off, seg, addr, pl, plsz, rw))
4672                         goto out;
4673                 while (*pl) {
4674                         ASSERT3U((*pl)->p_offset, ==, off);
4675                         off += PAGESIZE;
4676                         addr += PAGESIZE;
4677                         if (len > 0) {
4678                                 ASSERT3U(len, >=, PAGESIZE);
4679                                 len -= PAGESIZE;
4680                         }
4681                         ASSERT3U(plsz, >=, PAGESIZE);
```



```
4682                         plsz -= PAGESIZE;
4683                         pl++;
4684                 }
4685         }

4687         /*
4688          * Fill out the page array with any pages already in the cache.
4689          */
4690         while (plsz > 0 &&
4691             (*pl++ = page_lookup_nowait(vp, off, SE_SHARED))) {
4692                         off += PAGESIZE;
4693                         plsz -= PAGESIZE;
4694         }
4695 out:
4696         if (err) {
4697                 /*
4698                  * Release any pages we have previously locked.
4699                  */
4700                 while (pl > pl0)
4701                         page_unlock(*--pl);
4702         } else {
4703                 ZFS_ACCESSTIME_STAMP(zfsvfs, zp);
4704         }

4706         *pl = NULL;

4708         ZFS_EXIT(zfsvfs);
4709         return (err);
4710 }

4712 /*
4713  * Request a memory map for a section of a file.  This code interacts
4714  * with common code and the VM system as follows:
4715  *
4716  * - common code calls mmap(), which ends up in smmap_common()
4717  * - this calls VOP_MAP(), which takes you into (say) zfs
4718  * - zfs_map() calls as_map(), passing segvn_create() as the callback
4719  * - segvn_create() creates the new segment and calls VOP_ADDMAP()
4720  * - zfs_addmap() updates z_mapcnt
4721  */
4722 /*ARGSUSED*/
4723 static int
4724 zfs_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp,
4725     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
4726     caller_context_t *ct)
4727 {
4728         znode_t *zp = VTOZ(vp);
4729         zfsvfs_t *zfsvfs = zp->z_zfsvfs;
4730         segvn_crargs_t  vn_a;
4731         int             error;

4733         ZFS_ENTER(zfsvfs);
4734         ZFS_VERIFY_ZP(zp);

4736         if ((prot & PROT_WRITE) && (zp->z_pflags &
4737             (ZFS_IMMUTABLE | ZFS_READONLY | ZFS_APPENDONLY))) {
4738                 ZFS_EXIT(zfsvfs);
4739                 return (SET_ERROR(EPERM));
4740         }

4742         if ((prot & (PROT_READ | PROT_EXEC)) &&
4743             (zp->z_pflags & ZFS_AV_QUARANTINED)) {
4744                 ZFS_EXIT(zfsvfs);
4745                 return (SET_ERROR(EACCES));
4746         }
```

```
4748         if (vp->v_flag & VNOMAP) {
4749                 ZFS_EXIT(zfsvfs);
4750                 return (SET_ERROR(ENOSYS));
4751         }
4752
4753         if (off < 0 || len > MAXOFFSET_T - off) {
4754                 ZFS_EXIT(zfsvfs);
4755                 return (SET_ERROR(ENXIO));
4756         }
4757
4758         if (vp->v_type != VREG) {
4759                 ZFS_EXIT(zfsvfs);
4760                 return (SET_ERROR(ENODEV));
4761         }
4762
4763         /*
4764          * If file is locked, disallow mapping.
4765          */
4766         if (MANDMODE(zp->z_mode) && vn_has_flocks(vp)) {
4767                 ZFS_EXIT(zfsvfs);
4768                 return (SET_ERROR(EAGAIN));
4769         }
4770
4771         as_rangelock(as);
4772         error = choose_addr(as, addrp, len, off, ADDR_VACALIGN, flags);
4773         if (error != 0) {
4774                 as_rangeunlock(as);
4775                 ZFS_EXIT(zfsvfs);
4776                 return (error);
4777         }
4778
4779         vn_a.vp = vp;
4780         vn_a.offset = (u_offset_t)off;
4781         vn_a.type = flags & MAP_TYPE;
4782         vn_a.prot = prot;
4783         vn_a.maxprot = maxprot;
4784         vn_a.cred = cr;
4785         vn_a.amp = NULL;
4786         vn_a.flags = flags & ~MAP_TYPE;
4787         vn_a.szc = 0;
4788         vn_a.lgrp_mem_policy_flags = 0;
4789
4790         error = as_map(as, *addrp, len, segvn_create, &vn_a);
4791
4792         as_rangeunlock(as);
4793         ZFS_EXIT(zfsvfs);
4794         return (error);
4795 }
4796
4797 /* ARGSUSED */
4798 static int
4799 zfs_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
4800     size_t len, uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr,
4801     caller_context_t *ct)
4802 {
4803         uint64_t pages = btopr(len);
4804
4805         atomic_add_64(&VTOZ(vp)->z_mapcnt, pages);
4806         return (0);
4807 }
4808
4809 /*
4810  * The reason we push dirty pages as part of zfs_delmap() is so that we get a
4811  * more accurate mtime for the associated file.  Since we don't have a way of
4812  * detecting when the data was actually modified, we have to resort to
4813  * heuristics.  If an explicit msync() is done, then we mark the mtime when the
```



```
4814  * last page is pushed.  The problem occurs when the msync() call is omitted,
4815  * which by far the most common case:
4816  *
4817  *      open()
4818  *      mmap()
4819  *      <modify memory>
4820  *      munmap()
4821  *      close()
4822  *      <time lapse>
4823  *      putpage() via fsflush
4824  *
4825  * If we wait until fsflush to come along, we can have a modification time that
4826  * is some arbitrary point in the future.  In order to prevent this in the
4827  * common case, we flush pages whenever a (MAP_SHARED, PROT_WRITE) mapping is
4828  * torn down.
4829  */
4830 /* ARGSUSED */
4831 static int
4832 zfs_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
4833     size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr,
4834     caller_context_t *ct)
4835 {
4836         uint64_t pages = btopr(len);
4837
4838         ASSERT3U(VTOZ(vp)->z_mapcnt, >=, pages);
4839         atomic_add_64(&VTOZ(vp)->z_mapcnt, -pages);
4840
4841         if ((flags & MAP_SHARED) && (prot & PROT_WRITE) &&
4842             vn_has_cached_data(vp))
4843                 (void) VOP_PUTPAGE(vp, off, len, B_ASYNC, cr, ct);
4844
4845         return (0);
4846 }
4847
4848 /*
4849  * Free or allocate space in a file.  Currently, this function only
4850  * supports the 'F_FREESP' command.  However, this command is somewhat
4851  * misnamed, as its functionality includes the ability to allocate as
4852  * well as free space.
4853  *
4854  *      IN:     vp      - vnode of file to free data in.
4855  *              cmd     - action to take (only F_FREESP supported).
4856  *              bfp     - section of file to free/alloc.
4857  *              flag    - current file open mode flags.
4858  *              offset  - current file offset.
4859  *              cr      - credentials of caller [UNUSED].
4860  *              ct      - caller context.
4861  *
4862  *      RETURN: 0 on success, error code on failure.
4863  *
4864  * Timestamps:
4865  *      vp - ctime|mtime updated
4866  */
4867 /* ARGSUSED */
4868 static int
4869 zfs_space(vnode_t *vp, int cmd, flock64_t *bfp, int flag,
4870     offset_t offset, cred_t *cr, caller_context_t *ct)
4871 {
4872         znode_t         *zp = VTOZ(vp);
4873         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4874         uint64_t        off, len;
4875         int             error;
4876
4877         ZFS_ENTER(zfsvfs);
4878         ZFS_VERIFY_ZP(zp);
```

```
4880         if (cmd != F_FREESP) {
4881                 ZFS_EXIT(zfsvfs);
4882                 return (SET_ERROR(EINVAL));
4883         }
4885         /*
4886          * In a case vp->v_vfsp != zp->z_zfsvfs->z_vfs (e.g. snapshots) our
4887          * callers might not be able to detect properly that we are read-only,
4888          * so check it explicitly here.
4889          */
4890         if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
4891                 ZFS_EXIT(zfsvfs);
4892                 return (SET_ERROR(EROFS));
4893         }
4895         if (error = convoff(vp, bfp, 0, offset)) {
4896                 ZFS_EXIT(zfsvfs);
4897                 return (error);
4898         }
4900         if (bfp->l_len < 0) {
4901                 ZFS_EXIT(zfsvfs);
4902                 return (SET_ERROR(EINVAL));
4903         }
4905         off = bfp->l_start;
4906         len = bfp->l_len; /* 0 means from off to end of file */
4908         error = zfs_freesp(zp, off, len, flag, TRUE);
4910         if (error == 0 && off == 0 && len == 0)
4911                 vnevent_truncate(ZTOV(zp), ct);
4913         ZFS_EXIT(zfsvfs);
4914         return (error);
4915 }
4917 /*ARGSUSED*/
4918 static int
4919 zfs_fid(vnode_t *vp, fid_t *fidp, caller_context_t *ct)
4920 {
4921         znode_t         *zp = VTOZ(vp);
4922         zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
4923         uint32_t        gen;
4924         uint64_t        gen64;
4925         uint64_t        object = zp->z_id;
4926         zfid_short_t    *zfid;
4927         int             size, i, error;
4929         ZFS_ENTER(zfsvfs);
4930         ZFS_VERIFY_ZP(zp);
4932         if ((error = sa_lookup(zp->z_sa_hdl, SA_ZPL_GEN(zfsvfs),
4933             &gen64, sizeof (uint64_t))) != 0) {
4934                 ZFS_EXIT(zfsvfs);
4935                 return (error);
4936         }
4938         gen = (uint32_t)gen64;
4940         size = (zfsvfs->z_parent != zfsvfs) ? LONG_FID_LEN : SHORT_FID_LEN;
4941         if (fidp->fid_len < size) {
4942                 fidp->fid_len = size;
4943                 ZFS_EXIT(zfsvfs);
4944                 return (SET_ERROR(ENOSPC));
4945         }
```



```
4947         zfid = (zfid_short_t *)fidp;
4949         zfid->zf_len = size;
4951         for (i = 0; i < sizeof (zfid->zf_object); i++)
4952                 zfid->zf_object[i] = (uint8_t)(object >> (8 * i));
4954         /* Must have a non-zero generation number to distinguish from .zfs */
4955         if (gen == 0)
4956                 gen = 1;
4957         for (i = 0; i < sizeof (zfid->zf_gen); i++)
4958                 zfid->zf_gen[i] = (uint8_t)(gen >> (8 * i));
4960         if (size == LONG_FID_LEN) {
4961                 uint64_t        objsetid = dmu_objset_id(zfsvfs->z_os);
4962                 zfid_long_t     *zlfid;
4964                 zlfid = (zfid_long_t *)fidp;
4966                 for (i = 0; i < sizeof (zlfid->zf_setid); i++)
4967                         zlfid->zf_setid[i] = (uint8_t)(objsetid >> (8 * i));
4969                 /* XXX - this should be the generation number for the objset */
4970                 for (i = 0; i < sizeof (zlfid->zf_setgen); i++)
4971                         zlfid->zf_setgen[i] = 0;
4972         }
4974         ZFS_EXIT(zfsvfs);
4975         return (0);
4976 }
4978 static int
4979 zfs_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr,
4980     caller_context_t *ct)
4981 {
4982         znode_t         *zp, *xzp;
4983         zfsvfs_t        *zfsvfs;
4984         zfs_dirlock_t   *dl;
4985         int             error;
4987         switch (cmd) {
4988         case _PC_LINK_MAX:
4989                 *valp = ULONG_MAX;
4990                 return (0);
4992         case _PC_FILESIZEBITS:
4993                 *valp = 64;
4994                 return (0);
4996         case _PC_XATTR_EXISTS:
4997                 zp = VTOZ(vp);
4998                 zfsvfs = zp->z_zfsvfs;
4999                 ZFS_ENTER(zfsvfs);
5000                 ZFS_VERIFY_ZP(zp);
5001                 *valp = 0;
5002                 error = zfs_dirent_lock(&dl, zp, "", &xzp,
5003                     ZXATTR | ZEXISTS | ZSHARED, NULL, NULL);
5004                 if (error == 0) {
5005                         zfs_dirent_unlock(dl);
5006                         if (!zfs_dirempty(xzp))
5007                                 *valp = 1;
5008                         VN_RELE(ZTOV(xzp));
5009                 } else if (error == ENOENT) {
5010                         /*
5011                          * If there aren't extended attributes, it's the
```

```
5012                         * same as having zero of them.
5013                         */
5014                        error = 0;
5015                }
5016                ZFS_EXIT(zfsvfs);
5017                return (error);
5018
5019        case _PC_SATTR_ENABLED:
5020        case _PC_SATTR_EXISTS:
5021                *valp = vfs_has_feature(vp->v_vfsp, VFSFT_SYSATTR_VIEWS) &&
5022                    (vp->v_type == VREG || vp->v_type == VDIR);
5023                return (0);
5024
5025        case _PC_ACCESS_FILTERING:
5026                *valp = vfs_has_feature(vp->v_vfsp, VFSFT_ACCESS_FILTER) &&
5027                    vp->v_type == VDIR;
5028                return (0);
5029
5030        case _PC_ACL_ENABLED:
5031                *valp = _ACL_ACE_ENABLED;
5032                return (0);
5033
5034        case _PC_MIN_HOLE_SIZE:
5035                *valp = (ulong_t)SPA_MINBLOCKSIZE;
5036                return (0);
5037
5038        case _PC_TIMESTAMP_RESOLUTION:
5039                /* nanosecond timestamp resolution */
5040                *valp = 1L;
5041                return (0);
5042
5043        default:
5044                return (fs_pathconf(vp, cmd, valp, cr, ct));
5045        }
5046 }
5047
5048 /*ARGSUSED*/
5049 static int
5050 zfs_getsecattr(vnode_t *vp, vsecattr_t *vsecp, int flag, cred_t *cr,
5051     caller_context_t *ct)
5052 {
5053        znode_t *zp = VTOZ(vp);
5054        zfsvfs_t *zfsvfs = zp->z_zfsvfs;
5055        int error;
5056        boolean_t skipaclchk = (flag & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
5057
5058        ZFS_ENTER(zfsvfs);
5059        ZFS_VERIFY_ZP(zp);
5060        error = zfs_getacl(zp, vsecp, skipaclchk, cr);
5061        ZFS_EXIT(zfsvfs);
5062
5063        return (error);
5064 }
5065
5066 /*ARGSUSED*/
5067 static int
5068 zfs_setsecattr(vnode_t *vp, vsecattr_t *vsecp, int flag, cred_t *cr,
5069     caller_context_t *ct)
5070 {
5071        znode_t *zp = VTOZ(vp);
5072        zfsvfs_t *zfsvfs = zp->z_zfsvfs;
5073        int error;
5074        boolean_t skipaclchk = (flag & ATTR_NOACLCHECK) ? B_TRUE : B_FALSE;
5075        zilog_t *zilog = zfsvfs->z_log;
5076
5077        ZFS_ENTER(zfsvfs);
```



```
5078        ZFS_VERIFY_ZP(zp);
5079
5080        error = zfs_setacl(zp, vsecp, skipaclchk, cr);
5081
5082        if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
5083                zil_commit(zilog, 0);
5084
5085        ZFS_EXIT(zfsvfs);
5086        return (error);
5087 }
5088
5089 /*
5090  * The smallest read we may consider to loan out an arcbuf.
5091  * This must be a power of 2.
5092  */
5093 int zcr_blksz_min = (1 << 10);  /* 1K */
5094 /*
5095  * If set to less than the file block size, allow loaning out of an
5096  * arcbuf for a partial block read.  This must be a power of 2.
5097  */
5098 int zcr_blksz_max = (1 << 17);  /* 128K */
5099
5100 /*ARGSUSED*/
5101 static int
5102 zfs_reqzcbuf(vnode_t *vp, enum uio_rw ioflag, xuio_t *xuio, cred_t *cr,
5103     caller_context_t *ct)
5104 {
5105        znode_t *zp = VTOZ(vp);
5106        zfsvfs_t *zfsvfs = zp->z_zfsvfs;
5107        int max_blksz = zfsvfs->z_max_blksz;
5108        uio_t *uio = &xuio->xu_uio;
5109        ssize_t size = uio->uio_resid;
5110        offset_t offset = uio->uio_loffset;
5111        int blksz;
5112        int fullblk, i;
5113        arc_buf_t *abuf;
5114        ssize_t maxsize;
5115        int preamble, postamble;
5116
5117        if (xuio->xu_type != UIOTYPE_ZEROCOPY)
5118                return (SET_ERROR(EINVAL));
5119
5120        ZFS_ENTER(zfsvfs);
5121        ZFS_VERIFY_ZP(zp);
5122        switch (ioflag) {
5123        case UIO_WRITE:
5124                /*
5125                 * Loan out an arc_buf for write if write size is bigger than
5126                 * max_blksz, and the file's block size is also max_blksz.
5127                 */
5128                blksz = max_blksz;
5129                if (size < blksz || zp->z_blksz != blksz) {
5130                        ZFS_EXIT(zfsvfs);
5131                        return (SET_ERROR(EINVAL));
5132                }
5133                /*
5134                 * Caller requests buffers for write before knowing where the
5135                 * write offset might be (e.g. NFS TCP write).
5136                 */
5137                if (offset == -1) {
5138                        preamble = 0;
5139                } else {
5140                        preamble = P2PHASE(offset, blksz);
5141                        if (preamble) {
5142                                preamble = blksz - preamble;
5143                                size -= preamble;
```

```
5144                    }
5145                }
5146
5147                postamble = P2PHASE(size, blksz);
5148                size -= postamble;
5149
5150                fullblk = size / blksz;
5151                (void) dmu_xuio_init(xuio,
5152                    (preamble != 0) + fullblk + (postamble != 0));
5153                DTRACE_PROBE3(zfs_reqzcbuf_align, int, preamble,
5154                    int, postamble, int,
5155                    (preamble != 0) + fullblk + (postamble != 0));
5156
5157                /*
5158                 * Have to fix iov base/len for partial buffers.  They
5159                 * currently represent full arc_buf's.
5160                 */
5161                if (preamble) {
5162                        /* data begins in the middle of the arc_buf */
5163                        abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5164                            blksz);
5165                        ASSERT(abuf);
5166                        (void) dmu_xuio_add(xuio, abuf,
5167                            blksz - preamble, preamble);
5168                }
5169
5170                for (i = 0; i < fullblk; i++) {
5171                        abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5172                            blksz);
5173                        ASSERT(abuf);
5174                        (void) dmu_xuio_add(xuio, abuf, 0, blksz);
5175                }
5176
5177                if (postamble) {
5178                        /* data ends in the middle of the arc_buf */
5179                        abuf = dmu_request_arcbuf(sa_get_db(zp->z_sa_hdl),
5180                            blksz);
5181                        ASSERT(abuf);
5182                        (void) dmu_xuio_add(xuio, abuf, 0, postamble);
5183                }
5184                break;
5185        case UIO_READ:
5186                /*
5187                 * Loan out an arc_buf for read if the read size is larger than
5188                 * the current file block size.  Block alignment is not
5189                 * considered.  Partial arc_buf will be loaned out for read.
5190                 */
5191                blksz = zp->z_blksz;
5192                if (blksz < zcr_blksz_min)
5193                        blksz = zcr_blksz_min;
5194                if (blksz > zcr_blksz_max)
5195                        blksz = zcr_blksz_max;
5196                /* avoid potential complexity of dealing with it */
5197                if (blksz > max_blksz) {
5198                        ZFS_EXIT(zfsvfs);
5199                        return (SET_ERROR(EINVAL));
5200                }
5201
5202                maxsize = zp->z_size - uio->uio_loffset;
5203                if (size > maxsize)
5204                        size = maxsize;
5205
5206                if (size < blksz || vn_has_cached_data(vp)) {
5207                        ZFS_EXIT(zfsvfs);
5208                        return (SET_ERROR(EINVAL));
5209                }
```



```
5210                break;
5211        default:
5212                ZFS_EXIT(zfsvfs);
5213                return (SET_ERROR(EINVAL));
5214        }
5215
5216        uio->uio_extflg = UIO_XUIO;
5217        XUIO_XUZC_RW(xuio) = ioflag;
5218        ZFS_EXIT(zfsvfs);
5219        return (0);
5220 }
5221
5222 /*ARGSUSED*/
5223 static int
5224 zfs_retzcbuf(vnode_t *vp, xuio_t *xuio, cred_t *cr, caller_context_t *ct)
5225 {
5226        int i;
5227        arc_buf_t *abuf;
5228        int ioflag = XUIO_XUZC_RW(xuio);
5229
5230        ASSERT(xuio->xu_type == UIOTYPE_ZEROCOPY);
5231
5232        i = dmu_xuio_cnt(xuio);
5233        while (i-- > 0) {
5234                abuf = dmu_xuio_arcbuf(xuio, i);
5235                /*
5236                 * if abuf == NULL, it must be a write buffer
5237                 * that has been returned in zfs_write().
5238                 */
5239                if (abuf)
5240                        dmu_return_arcbuf(abuf);
5241                ASSERT(abuf || ioflag == UIO_WRITE);
5242        }
5243
5244        dmu_xuio_fini(xuio);
5245        return (0);
5246 }
5247
5248 /*
5249  * Predeclare these here so that the compiler assumes that
5250  * this is an "old style" function declaration that does
5251  * not include arguments => we won't get type mismatch errors
5252  * in the initializations that follow.
5253  */
5254 static int zfs_inval();
5255 static int zfs_isdir();
5256
5257 static int
5258 zfs_inval()
5259 {
5260        return (SET_ERROR(EINVAL));
5261 }
5262
5263 static int
5264 zfs_isdir()
5265 {
5266        return (SET_ERROR(EISDIR));
5267 }
5268 /*
5269  * Directory vnode operations template
5270  */
5271 vnodeops_t *zfs_dvnodeops;
5272 const fs_operation_def_t zfs_dvnodeops_template[] = {
5273        VOPNAME_OPEN,           { .vop_open = zfs_open },
5274        VOPNAME_CLOSE,          { .vop_close = zfs_close },
5275        VOPNAME_READ,           { .error = zfs_isdir },
```

```
5276         VOPNAME_WRITE,          { .error = zfs_isdir },
5277         VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5278         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5279         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5280         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5281         VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5282         VOPNAME_CREATE,         { .vop_create = zfs_create },
5283         VOPNAME_REMOVE,         { .vop_remove = zfs_remove },
5284         VOPNAME_LINK,           { .vop_link = zfs_link },
5285         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5286         VOPNAME_MKDIR,          { .vop_mkdir = zfs_mkdir },
5287         VOPNAME_RMDIR,          { .vop_rmdir = zfs_rmdir },
5288         VOPNAME_READDIR,        { .vop_readdir = zfs_readdir },
5289         VOPNAME_SYMLINK,        { .vop_symlink = zfs_symlink },
5290         VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5291         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5292         VOPNAME_FID,            { .vop_fid = zfs_fid },
5293         VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5294         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5295         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5296         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5297         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5298         NULL,                   NULL
5299 };

5301 /*
5302  * Regular file vnode operations template
5303  */
5304 vnodeops_t *zfs_fvnodeops;
5305 const fs_operation_def_t zfs_fvnodeops_template[] = {
5306         VOPNAME_OPEN,           { .vop_open = zfs_open },
5307         VOPNAME_CLOSE,          { .vop_close = zfs_close },
5308         VOPNAME_READ,           { .vop_read = zfs_read },
5309         VOPNAME_WRITE,          { .vop_write = zfs_write },
5310         VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5311         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5312         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5313         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5314         VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5315         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5316         VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5317         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5318         VOPNAME_FID,            { .vop_fid = zfs_fid },
5319         VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5320         VOPNAME_FRLOCK,         { .vop_frlock = zfs_frlock },
5321         VOPNAME_SPACE,          { .vop_space = zfs_space },
5322         VOPNAME_GETPAGE,        { .vop_getpage = zfs_getpage },
5323         VOPNAME_PUTPAGE,        { .vop_putpage = zfs_putpage },
5324         VOPNAME_MAP,            { .vop_map = zfs_map },
5325         VOPNAME_ADDMAP,         { .vop_addmap = zfs_addmap },
5326         VOPNAME_DELMAP,         { .vop_delmap = zfs_delmap },
5327         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5328         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5329         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5330         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5331         VOPNAME_REQZCBUF,       { .vop_reqzcbuf = zfs_reqzcbuf },
5332         VOPNAME_RETZCBUF,       { .vop_retzcbuf = zfs_retzcbuf },
5333         NULL,                   NULL
5334 };

5336 /*
5337  * Symbolic link vnode operations template
5338  */
5339 vnodeops_t *zfs_symvnodeops;
5340 const fs_operation_def_t zfs_symvnodeops_template[] = {
5341         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
```



```
5342         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5343         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5344         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5345         VOPNAME_READLINK,       { .vop_readlink = zfs_readlink },
5346         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5347         VOPNAME_FID,            { .vop_fid = zfs_fid },
5348         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5349         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5350         NULL,                   NULL
5351 };

5353 /*
5354  * special share hidden files vnode operations template
5355  */
5356 vnodeops_t *zfs_sharevnodeops;
5357 const fs_operation_def_t zfs_sharevnodeops_template[] = {
5358         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5359         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5360         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5361         VOPNAME_FID,            { .vop_fid = zfs_fid },
5362         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5363         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5364         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5365         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5366         NULL,                   NULL
5367 };

5369 /*
5370  * Extended attribute directory vnode operations template
5371  *
5372  * This template is identical to the directory vnodes
5373  * operation template except for restricted operations:
5374  *      VOP_MKDIR()
5375  *      VOP_SYMLINK()
5376  *
5377  * Note that there are other restrictions embedded in:
5378  *      zfs_create()    - restrict type to VREG
5379  *      zfs_link()      - no links into/out of attribute space
5380  *      zfs_rename()    - no moves into/out of attribute space
5381  */
5382 vnodeops_t *zfs_xdvnodeops;
5383 const fs_operation_def_t zfs_xdvnodeops_template[] = {
5384         VOPNAME_OPEN,           { .vop_open = zfs_open },
5385         VOPNAME_CLOSE,          { .vop_close = zfs_close },
5386         VOPNAME_IOCTL,          { .vop_ioctl = zfs_ioctl },
5387         VOPNAME_GETATTR,        { .vop_getattr = zfs_getattr },
5388         VOPNAME_SETATTR,        { .vop_setattr = zfs_setattr },
5389         VOPNAME_ACCESS,         { .vop_access = zfs_access },
5390         VOPNAME_LOOKUP,         { .vop_lookup = zfs_lookup },
5391         VOPNAME_CREATE,         { .vop_create = zfs_create },
5392         VOPNAME_REMOVE,         { .vop_remove = zfs_remove },
5393         VOPNAME_LINK,           { .vop_link = zfs_link },
5394         VOPNAME_RENAME,         { .vop_rename = zfs_rename },
5395         VOPNAME_MKDIR,          { .error = zfs_inval },
5396         VOPNAME_RMDIR,          { .vop_rmdir = zfs_rmdir },
5397         VOPNAME_READDIR,        { .vop_readdir = zfs_readdir },
5398         VOPNAME_SYMLINK,        { .error = zfs_inval },
5399         VOPNAME_FSYNC,          { .vop_fsync = zfs_fsync },
5400         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5401         VOPNAME_FID,            { .vop_fid = zfs_fid },
5402         VOPNAME_SEEK,           { .vop_seek = zfs_seek },
5403         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5404         VOPNAME_GETSECATTR,     { .vop_getsecattr = zfs_getsecattr },
5405         VOPNAME_SETSECATTR,     { .vop_setsecattr = zfs_setsecattr },
5406         VOPNAME_VNEVENT,        { .vop_vnevent = fs_vnevent_support },
5407         NULL,                   NULL
```

```
5408 };

5410 /*
5411  * Error vnode operations template
5412  */
5413 vnodeops_t *zfs_evnodeops;
5414 const fs_operation_def_t zfs_evnodeops_template[] = {
5415         VOPNAME_INACTIVE,       { .vop_inactive = zfs_inactive },
5416         VOPNAME_PATHCONF,       { .vop_pathconf = zfs_pathconf },
5417         NULL,                   NULL
5418 };
```

```
**********************************************************
   93682 Tue Oct 28 11:57:20 2014
new/usr/src/uts/common/fs/zfs/zio.c
Possibility to physically reserve space without writing leaf blocks
**********************************************************
______unchanged_portion_omitted_

 950 /*
 951  * ==========================================================================
 952  * Prepare to read and write logical blocks
 953  * ==========================================================================
 954  */

 956 static int
 957 zio_read_bp_init(zio_t *zio)
 958 {
 959         blkptr_t *bp = zio->io_bp;

 961         if (!BP_IS_EMBEDDED(bp) && BP_GET_PROP_RESERVATION(bp)) {
 962                 memset(zio->io_orig_data, 0, zio->io_orig_size);
 963                 zio->io_pipeline = ZIO_INTERLOCK_STAGES;
 964                 return (ZIO_PIPELINE_CONTINUE);
 965         }

 967 #endif /* ! codereview */
 968         if (BP_GET_COMPRESS(bp) != ZIO_COMPRESS_OFF &&
 969             zio->io_child_type == ZIO_CHILD_LOGICAL &&
 970             !(zio->io_flags & ZIO_FLAG_RAW)) {
 971                 uint64_t psize =
 972                     BP_IS_EMBEDDED(bp) ? BPE_GET_PSIZE(bp) : BP_GET_PSIZE(bp);
 973                 void *cbuf = zio_buf_alloc(psize);

 975                 zio_push_transform(zio, cbuf, psize, psize, zio_decompress);
 976         }

 978         if (BP_IS_EMBEDDED(bp) && BPE_GET_ETYPE(bp) == BP_EMBEDDED_TYPE_DATA) {
 979                 zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;
 980                 decode_embedded_bp_compressed(bp, zio->io_data);
 981         } else {
 982                 ASSERT(!BP_IS_EMBEDDED(bp));
 983         }

 985         if (!DMU_OT_IS_METADATA(BP_GET_TYPE(bp)) && BP_GET_LEVEL(bp) == 0)
 986                 zio->io_flags |= ZIO_FLAG_DONT_CACHE;

 988         if (BP_GET_TYPE(bp) == DMU_OT_DDT_ZAP)
 989                 zio->io_flags |= ZIO_FLAG_DONT_CACHE;

 991         if (BP_GET_DEDUP(bp) && zio->io_child_type == ZIO_CHILD_LOGICAL)
 992                 zio->io_pipeline = ZIO_DDT_READ_PIPELINE;

 994         return (ZIO_PIPELINE_CONTINUE);
 995 }

 997 static int
 998 zio_write_bp_init(zio_t *zio)
 999 {
1000         spa_t *spa = zio->io_spa;
1001         zio_prop_t *zp = &zio->io_prop;
1002         enum zio_compress compress = zp->zp_compress;
1003         enum zio_checksum checksum = zp->zp_checksum;
1004         uint8_t dedup = zp->zp_dedup;
1005 #endif /* ! codereview */
1006         blkptr_t *bp = zio->io_bp;
1007         uint64_t lsize = zio->io_size;
1008         uint64_t psize = lsize;
```



```
1009         int pass = 1;

1011         /*
1012          * If our children haven't all reached the ready stage,
1013          * wait for them and then repeat this pipeline stage.
1014          */
1015         if (zio_wait_for_children(zio, ZIO_CHILD_GANG, ZIO_WAIT_READY) ||
1016             zio_wait_for_children(zio, ZIO_CHILD_LOGICAL, ZIO_WAIT_READY))
1017                 return (ZIO_PIPELINE_STOP);

1019         if (!IO_IS_ALLOCATING(zio))
1020                 return (ZIO_PIPELINE_CONTINUE);

1022         ASSERT(zio->io_child_type != ZIO_CHILD_DDT);

1024         if (zp->zp_zero_write && !(zio->io_pipeline & ZIO_GANG_STAGES)) {
1025                 dedup = B_FALSE;
1026                 compress = ZIO_COMPRESS_OFF;
1027                 checksum = ZIO_CHECKSUM_OFF;
1028         }

1030 #endif /* ! codereview */
1031         if (zio->io_bp_override) {
1032                 ASSERT(bp->blk_birth != zio->io_txg);
1033                 ASSERT(BP_GET_DEDUP(zio->io_bp_override) == 0);

1035                 *bp = *zio->io_bp_override;
1036                 zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;

1038                 if (BP_IS_EMBEDDED(bp))
1039                         return (ZIO_PIPELINE_CONTINUE);

1041                 /*
1042                  * If we've been overridden and nopwrite is set then
1043                  * set the flag accordingly to indicate that a nopwrite
1044                  * has already occurred.
1045                  */
1046                 if (!BP_IS_HOLE(bp) && zp->zp_nopwrite) {
1047                         ASSERT(!zp->zp_dedup);
1048                         zio->io_flags |= ZIO_FLAG_NOPWRITE;
1049                         return (ZIO_PIPELINE_CONTINUE);
1050                 }

1052                 ASSERT(!zp->zp_nopwrite);

1054                 if (BP_IS_HOLE(bp) || !dedup)
 961                 if (BP_IS_HOLE(bp) || !zp->zp_dedup)
1055                         return (ZIO_PIPELINE_CONTINUE);

1057                 ASSERT(zio_checksum_table[checksum].ci_dedup ||
 964                 ASSERT(zio_checksum_table[zp->zp_checksum].ci_dedup ||
1058                     zp->zp_dedup_verify);

1060                 if (BP_GET_CHECKSUM(bp) == checksum) {
 967                 if (BP_GET_CHECKSUM(bp) == zp->zp_checksum) {
1061                         BP_SET_DEDUP(bp, 1);
1062                         zio->io_pipeline |= ZIO_STAGE_DDT_WRITE;
1063                         return (ZIO_PIPELINE_CONTINUE);
1064                 }
1065                 zio->io_bp_override = NULL;
1066                 BP_ZERO(bp);
1067         }

1069         if (!BP_IS_HOLE(bp) && bp->blk_birth == zio->io_txg) {
1070                 /*
1071                  * We're rewriting an existing block, which means we're
```

```
1072                 * working on behalf of spa_sync().  For spa_sync() to
1073                 * converge, it must eventually be the case that we don't
1074                 * have to allocate new blocks.  But compression changes
1075                 * the blocksize, which forces a reallocate, and makes
1076                 * convergence take longer.  Therefore, after the first
1077                 * few passes, stop compressing to ensure convergence.
1078                 */
1079                pass = spa_sync_pass(spa);

1081                ASSERT(zio->io_txg == spa_syncing_txg(spa));
1082                ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);
1083                ASSERT(!BP_GET_DEDUP(bp));

1085                if (pass >= zfs_sync_pass_dont_compress)
1086                        compress = ZIO_COMPRESS_OFF;

1088                /* Make sure someone doesn't change their mind on overwrites */
1089                ASSERT(BP_IS_EMBEDDED(bp) || MIN(zp->zp_copies + BP_IS_GANG(bp),
1090                    spa_max_replication(spa)) == BP_GET_NDVAS(bp));
1091        }

1093        if (compress != ZIO_COMPRESS_OFF) {
1094                void *cbuf = zio_buf_alloc(lsize);
1095                psize = zio_compress_data(compress, zio->io_data, cbuf, lsize);
1096                if (psize == 0 || psize == lsize) {
1097                        compress = ZIO_COMPRESS_OFF;
1098                        zio_buf_free(cbuf, lsize);
1099                } else if (!zp->zp_dedup && psize <= BPE_PAYLOAD_SIZE &&
1100                    zp->zp_level == 0 && !DMU_OT_HAS_FILL(zp->zp_type) &&
1101                    spa_feature_is_enabled(spa, SPA_FEATURE_EMBEDDED_DATA)) {
1102                        encode_embedded_bp_compressed(bp,
1103                            cbuf, compress, lsize, psize);
1104                        BPE_SET_ETYPE(bp, BP_EMBEDDED_TYPE_DATA);
1105                        BP_SET_TYPE(bp, zio->io_prop.zp_type);
1106                        BP_SET_LEVEL(bp, zio->io_prop.zp_level);
1107                        zio_buf_free(cbuf, lsize);
1108                        bp->blk_birth = zio->io_txg;
1109                        zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;
1110                        ASSERT(spa_feature_is_active(spa,
1111                            SPA_FEATURE_EMBEDDED_DATA));
1112                        return (ZIO_PIPELINE_CONTINUE);
1113                } else {
1114                        /*
1115                         * Round up compressed size to MINBLOCKSIZE and
1116                         * zero the tail.
1117                         */
1118                        size_t rounded =
1119                            P2ROUNDUP(psize, (size_t)SPA_MINBLOCKSIZE);
1120                        if (rounded > psize) {
1121                                bzero((char *)cbuf + psize, rounded - psize);
1122                                psize = rounded;
1123                        }
1124                        if (psize == lsize) {
1125                                compress = ZIO_COMPRESS_OFF;
1126                                zio_buf_free(cbuf, lsize);
1127                        } else {
1128                                zio_push_transform(zio, cbuf,
1129                                    psize, lsize, NULL);
1130                        }
1131                }
1132        }

1134        /*
1135         * The final pass of spa_sync() must be all rewrites, but the first
1136         * few passes offer a trade-off: allocating blocks defers convergence,
1137         * but newly allocated blocks are sequential, so they can be written
```



```
1138         * to disk faster.  Therefore, we allow the first few passes of
1139         * spa_sync() to allocate new blocks, but force rewrites after that.
1140         * There should only be a handful of blocks after pass 1 in any case.
1141         */
1142        if (!BP_IS_HOLE(bp) && bp->blk_birth == zio->io_txg &&
1143            BP_GET_PSIZE(bp) == psize &&
1144            pass >= zfs_sync_pass_rewrite) {
1145                ASSERT(psize != 0);
1146                enum zio_stage gang_stages = zio->io_pipeline & ZIO_GANG_STAGES;
1147                zio->io_pipeline = ZIO_REWRITE_PIPELINE | gang_stages;
1148                zio->io_flags |= ZIO_FLAG_IO_REWRITE;
1149        } else {
1150                BP_ZERO(bp);
1151                zio->io_pipeline = ZIO_WRITE_PIPELINE;
1152        }

1154        if (psize == 0) {
1155                if (zio->io_bp_orig.blk_birth != 0 &&
1156                    spa_feature_is_active(spa, SPA_FEATURE_HOLE_BIRTH)) {
1157                        BP_SET_LSIZE(bp, lsize);
1158                        BP_SET_TYPE(bp, zp->zp_type);
1159                        BP_SET_LEVEL(bp, zp->zp_level);
1160                        BP_SET_BIRTH(bp, zio->io_txg, 0);
1161                }
1162                zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;
1163        } else {
1164                ASSERT(zp->zp_checksum != ZIO_CHECKSUM_GANG_HEADER);
1165                BP_SET_LSIZE(bp, lsize);
1166                BP_SET_TYPE(bp, zp->zp_type);
1167                BP_SET_LEVEL(bp, zp->zp_level);
1168                BP_SET_PSIZE(bp, psize);
1169                BP_SET_COMPRESS(bp, compress);
1170                BP_SET_CHECKSUM(bp, zp->zp_checksum);
1171                BP_SET_DEDUP(bp, zp->zp_dedup);
1172                BP_SET_BYTEORDER(bp, ZFS_HOST_BYTEORDER);
1173                if (zp->zp_zero_write  && !(zio->io_pipeline & ZIO_GANG_STAGES))
1174                        boolean_t need_allocate = B_FALSE;
1175                        if (zio->io_pipeline & ZIO_STAGE_DVA_ALLOCATE)
1176                                need_allocate = B_TRUE;
1177                        zio->io_pipeline = ZIO_INTERLOCK_STAGES;
1178                        if (need_allocate)
1179                                zio->io_pipeline |= ZIO_STAGE_DVA_ALLOCATE;
1180                        BP_SET_PROP_RESERVATION(bp, 1);
1181                } else {
1182                        BP_SET_PROP_RESERVATION(bp, 0);
1183                }
1184 #endif /* ! codereview */
1185                if (zp->zp_dedup) {
1186                        ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);
1187                        ASSERT(!(zio->io_flags & ZIO_FLAG_IO_REWRITE));
1188                        zio->io_pipeline = ZIO_DDT_WRITE_PIPELINE;
1189                }
1190                if (zp->zp_nopwrite) {
1191                        ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);
1192                        ASSERT(!(zio->io_flags & ZIO_FLAG_IO_REWRITE));
1193                        zio->io_pipeline |= ZIO_STAGE_NOP_WRITE;
1194                }
1195        }

1197        return (ZIO_PIPELINE_CONTINUE);
1198 }

1200 static int
1201 zio_free_bp_init(zio_t *zio)
1202 {
1203         blkptr_t *bp = zio->io_bp;
```

```
1205         if (zio->io_child_type == ZIO_CHILD_LOGICAL) {
1206                 if (BP_GET_DEDUP(bp))
1207                         zio->io_pipeline = ZIO_DDT_FREE_PIPELINE;
1208         }
1209
1210         return (ZIO_PIPELINE_CONTINUE);
1211 }
1212
1213 /*
1214  * ==========================================================================
1215  * Execute the I/O pipeline
1216  * ==========================================================================
1217  */
1218
1219 static void
1220 zio_taskq_dispatch(zio_t *zio, zio_taskq_type_t q, boolean_t cutinline)
1221 {
1222         spa_t *spa = zio->io_spa;
1223         zio_type_t t = zio->io_type;
1224         int flags = (cutinline ? TQ_FRONT : 0);
1225
1226         /*
1227          * If we're a config writer or a probe, the normal issue and
1228          * interrupt threads may all be blocked waiting for the config lock.
1229          * In this case, select the otherwise-unused taskq for ZIO_TYPE_NULL.
1230          */
1231         if (zio->io_flags & (ZIO_FLAG_CONFIG_WRITER | ZIO_FLAG_PROBE))
1232                 t = ZIO_TYPE_NULL;
1233
1234         /*
1235          * A similar issue exists for the L2ARC write thread until L2ARC 2.0.
1236          */
1237         if (t == ZIO_TYPE_WRITE && zio->io_vd && zio->io_vd->vdev_aux)
1238                 t = ZIO_TYPE_NULL;
1239
1240         /*
1241          * If this is a high priority I/O, then use the high priority taskq if
1242          * available.
1243          */
1244         if (zio->io_priority == ZIO_PRIORITY_NOW &&
1245             spa->spa_zio_taskq[t][q + 1].stqs_count != 0)
1246                 q++;
1247
1248         ASSERT3U(q, <, ZIO_TASKQ_TYPES);
1249
1250         /*
1251          * NB: We are assuming that the zio can only be dispatched
1252          * to a single taskq at a time.  It would be a grievous error
1253          * to dispatch the zio to another taskq at the same time.
1254          */
1255         ASSERT(zio->io_tqent.tqent_next == NULL);
1256         spa_taskq_dispatch_ent(spa, t, q, (task_func_t *)zio_execute, zio,
1257             flags, &zio->io_tqent);
1258 }
1259
1260 static boolean_t
1261 zio_taskq_member(zio_t *zio, zio_taskq_type_t q)
1262 {
1263         kthread_t *executor = zio->io_executor;
1264         spa_t *spa = zio->io_spa;
1265
1266         for (zio_type_t t = 0; t < ZIO_TYPES; t++) {
1267                 spa_taskqs_t *tqs = &spa->spa_zio_taskq[t][q];
1268                 uint_t i;
1269                 for (i = 0; i < tqs->stqs_count; i++) {
```



```
1270                         if (taskq_member(tqs->stqs_taskq[i], executor))
1271                                 return (B_TRUE);
1272                 }
1273         }
1274
1275         return (B_FALSE);
1276 }
1277
1278 static int
1279 zio_issue_async(zio_t *zio)
1280 {
1281         zio_taskq_dispatch(zio, ZIO_TASKQ_ISSUE, B_FALSE);
1282
1283         return (ZIO_PIPELINE_STOP);
1284 }
1285
1286 void
1287 zio_interrupt(zio_t *zio)
1288 {
1289         zio_taskq_dispatch(zio, ZIO_TASKQ_INTERRUPT, B_FALSE);
1290 }
1291
1292 /*
1293  * Execute the I/O pipeline until one of the following occurs:
1294  *
1295  *      (1) the I/O completes
1296  *      (2) the pipeline stalls waiting for dependent child I/Os
1297  *      (3) the I/O issues, so we're waiting for an I/O completion interrupt
1298  *      (4) the I/O is delegated by vdev-level caching or aggregation
1299  *      (5) the I/O is deferred due to vdev-level queueing
1300  *      (6) the I/O is handed off to another thread.
1301  *
1302  * In all cases, the pipeline stops whenever there's no CPU work; it never
1303  * burns a thread in cv_wait().
1304  *
1305  * There's no locking on io_stage because there's no legitimate way
1306  * for multiple threads to be attempting to process the same I/O.
1307  */
1308 static zio_pipe_stage_t *zio_pipeline[];
1309
1310 void
1311 zio_execute(zio_t *zio)
1312 {
1313         zio->io_executor = curthread;
1314
1315         while (zio->io_stage < ZIO_STAGE_DONE) {
1316                 enum zio_stage pipeline = zio->io_pipeline;
1317                 enum zio_stage stage = zio->io_stage;
1318                 int rv;
1319
1320                 ASSERT(!MUTEX_HELD(&zio->io_lock));
1321                 ASSERT(ISP2(stage));
1322                 ASSERT(zio->io_stall == NULL);
1323
1324                 do {
1325                         stage <<= 1;
1326                 } while ((stage & pipeline) == 0);
1327
1328                 ASSERT(stage <= ZIO_STAGE_DONE);
1329
1330                 /*
1331                  * If we are in interrupt context and this pipeline stage
1332                  * will grab a config lock that is held across I/O,
1333                  * or may wait for an I/O that needs an interrupt thread
1334                  * to complete, issue async to avoid deadlock.
1335                  *
```

```
1336                  * For VDEV_IO_START, we cut in line so that the io will
1337                  * be sent to disk promptly.
1338                  */
1339                 if ((stage & ZIO_BLOCKING_STAGES) && zio->io_vd == NULL &&
1340                     zio_taskq_member(zio, ZIO_TASKQ_INTERRUPT)) {
1341                         boolean_t cut = (stage == ZIO_STAGE_VDEV_IO_START) ?
1342                             zio_requeue_io_start_cut_in_line : B_FALSE;
1343                         zio_taskq_dispatch(zio, ZIO_TASKQ_ISSUE, cut);
1344                         return;
1345                 }
1347                 zio->io_stage = stage;
1348                 rv = zio_pipeline[highbit64(stage) - 1](zio);
1350                 if (rv == ZIO_PIPELINE_STOP)
1351                         return;
1353                 ASSERT(rv == ZIO_PIPELINE_CONTINUE);
1354         }
1355 }
1357 /*
1358  * ==========================================================================
1359  * Initiate I/O, either sync or async
1360  * ==========================================================================
1361  */
1362 int
1363 zio_wait(zio_t *zio)
1364 {
1365         int error;
1367         ASSERT(zio->io_stage == ZIO_STAGE_OPEN);
1368         ASSERT(zio->io_executor == NULL);
1370         zio->io_waiter = curthread;
1372         zio_execute(zio);
1374         mutex_enter(&zio->io_lock);
1375         while (zio->io_executor != NULL)
1376                 cv_wait(&zio->io_cv, &zio->io_lock);
1377         mutex_exit(&zio->io_lock);
1379         error = zio->io_error;
1380         zio_destroy(zio);
1382         return (error);
1383 }
1385 void
1386 zio_nowait(zio_t *zio)
1387 {
1388         ASSERT(zio->io_executor == NULL);
1390         if (zio->io_child_type == ZIO_CHILD_LOGICAL &&
1391             zio_unique_parent(zio) == NULL) {
1392                 /*
1393                  * This is a logical async I/O with no parent to wait for it.
1394                  * We add it to the spa_async_root_zio "Godfather" I/O which
1395                  * will ensure they complete prior to unloading the pool.
1396                  */
1397                 spa_t *spa = zio->io_spa;
1399                 zio_add_child(spa->spa_async_zio_root[CPU_SEQID], zio);
1400         }
```



```
1402         zio_execute(zio);
1403 }
1405 /*
1406  * ==========================================================================
1407  * Reexecute or suspend/resume failed I/O
1408  * ==========================================================================
1409  */
1411 static void
1412 zio_reexecute(zio_t *pio)
1413 {
1414         zio_t *cio, *cio_next;
1416         ASSERT(pio->io_child_type == ZIO_CHILD_LOGICAL);
1417         ASSERT(pio->io_orig_stage == ZIO_STAGE_OPEN);
1418         ASSERT(pio->io_gang_leader == NULL);
1419         ASSERT(pio->io_gang_tree == NULL);
1421         pio->io_flags = pio->io_orig_flags;
1422         pio->io_stage = pio->io_orig_stage;
1423         pio->io_pipeline = pio->io_orig_pipeline;
1424         pio->io_reexecute = 0;
1425         pio->io_flags |= ZIO_FLAG_REEXECUTED;
1426         pio->io_error = 0;
1427         for (int w = 0; w < ZIO_WAIT_TYPES; w++)
1428                 pio->io_state[w] = 0;
1429         for (int c = 0; c < ZIO_CHILD_TYPES; c++)
1430                 pio->io_child_error[c] = 0;
1432         if (IO_IS_ALLOCATING(pio))
1433                 BP_ZERO(pio->io_bp);
1435         /*
1436          * As we reexecute pio's children, new children could be created.
1437          * New children go to the head of pio's io_child_list, however,
1438          * so we will (correctly) not reexecute them.  The key is that
1439          * the remainder of pio's io_child_list, from 'cio_next' onward,
1440          * cannot be affected by any side effects of reexecuting 'cio'.
1441          */
1442         for (cio = zio_walk_children(pio); cio != NULL; cio = cio_next) {
1443                 cio_next = zio_walk_children(pio);
1444                 mutex_enter(&pio->io_lock);
1445                 for (int w = 0; w < ZIO_WAIT_TYPES; w++)
1446                         pio->io_children[cio->io_child_type][w]++;
1447                 mutex_exit(&pio->io_lock);
1448                 zio_reexecute(cio);
1449         }
1451         /*
1452          * Now that all children have been reexecuted, execute the parent.
1453          * We don't reexecute "The Godfather" I/O here as it's the
1454          * responsibility of the caller to wait on him.
1455          */
1456         if (!(pio->io_flags & ZIO_FLAG_GODFATHER))
1457                 zio_execute(pio);
1458 }
1460 void
1461 zio_suspend(spa_t *spa, zio_t *zio)
1462 {
1463         if (spa_get_failmode(spa) == ZIO_FAILURE_MODE_PANIC)
1464                 fm_panic("Pool '%s' has encountered an uncorrectable I/O "
1465                     "failure and the failure mode property for this pool "
1466                     "is set to panic.", spa_name(spa));
```

```
1468         zfs_ereport_post(FM_EREPORT_ZFS_IO_FAILURE, spa, NULL, NULL, 0, 0);

1470         mutex_enter(&spa->spa_suspend_lock);

1472         if (spa->spa_suspend_zio_root == NULL)
1473                 spa->spa_suspend_zio_root = zio_root(spa, NULL, NULL,
1474                     ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE |
1475                     ZIO_FLAG_GODFATHER);

1477         spa->spa_suspended = B_TRUE;

1479         if (zio != NULL) {
1480                 ASSERT(!(zio->io_flags & ZIO_FLAG_GODFATHER));
1481                 ASSERT(zio != spa->spa_suspend_zio_root);
1482                 ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);
1483                 ASSERT(zio_unique_parent(zio) == NULL);
1484                 ASSERT(zio->io_stage == ZIO_STAGE_DONE);
1485                 zio_add_child(spa->spa_suspend_zio_root, zio);
1486         }

1488         mutex_exit(&spa->spa_suspend_lock);
1489 }

1491 int
1492 zio_resume(spa_t *spa)
1493 {
1494         zio_t *pio;

1496         /*
1497          * Reexecute all previously suspended i/o.
1498          */
1499         mutex_enter(&spa->spa_suspend_lock);
1500         spa->spa_suspended = B_FALSE;
1501         cv_broadcast(&spa->spa_suspend_cv);
1502         pio = spa->spa_suspend_zio_root;
1503         spa->spa_suspend_zio_root = NULL;
1504         mutex_exit(&spa->spa_suspend_lock);

1506         if (pio == NULL)
1507                 return (0);

1509         zio_reexecute(pio);
1510         return (zio_wait(pio));
1511 }

1513 void
1514 zio_resume_wait(spa_t *spa)
1515 {
1516         mutex_enter(&spa->spa_suspend_lock);
1517         while (spa_suspended(spa))
1518                 cv_wait(&spa->spa_suspend_cv, &spa->spa_suspend_lock);
1519         mutex_exit(&spa->spa_suspend_lock);
1520 }

1522 /*
1523  * ==========================================================================
1524  * Gang blocks.
1525  *
1526  * A gang block is a collection of small blocks that looks to the DMU
1527  * like one large block.  When zio_dva_allocate() cannot find a block
1528  * of the requested size, due to either severe fragmentation or the pool
1529  * being nearly full, it calls zio_write_gang_block() to construct the
1530  * block from smaller fragments.
1531  *
1532  * A gang block consists of a gang header (zio_gbh_phys_t) and up to
1533  * three (SPA_GBH_NBLKPTRS) gang members.  The gang header is just like
```



```
1534  * an indirect block: it's an array of block pointers.  It consumes
1535  * only one sector and hence is allocatable regardless of fragmentation.
1536  * The gang header's bps point to its gang members, which hold the data.
1537  *
1538  * Gang blocks are self-checksumming, using the bp's <vdev, offset, txg>
1539  * as the verifier to ensure uniqueness of the SHA256 checksum.
1540  * Critically, the gang block bp's blk_cksum is the checksum of the data,
1541  * not the gang header.  This ensures that data block signatures (needed for
1542  * deduplication) are independent of how the block is physically stored.
1543  *
1544  * Gang blocks can be nested: a gang member may itself be a gang block.
1545  * Thus every gang block is a tree in which root and all interior nodes are
1546  * gang headers, and the leaves are normal blocks that contain user data.
1547  * The root of the gang tree is called the gang leader.
1548  *
1549  * To perform any operation (read, rewrite, free, claim) on a gang block,
1550  * zio_gang_assemble() first assembles the gang tree (minus data leaves)
1551  * in the io_gang_tree field of the original logical i/o by recursively
1552  * reading the gang leader and all gang headers below it.  This yields
1553  * an in-core tree containing the contents of every gang header and the
1554  * bps for every constituent of the gang block.
1555  *
1556  * With the gang tree now assembled, zio_gang_issue() just walks the gang tree
1557  * and invokes a callback on each bp.  To free a gang block, zio_gang_issue()
1558  * calls zio_free_gang() -- a trivial wrapper around zio_free() -- for each bp.
1559  * zio_claim_gang() provides a similarly trivial wrapper for zio_claim().
1560  * zio_read_gang() is a wrapper around zio_read() that omits reading gang
1561  * headers, since we already have those in io_gang_tree.  zio_rewrite_gang()
1562  * performs a zio_rewrite() of the data or, for gang headers, a zio_rewrite()
1563  * of the gang header plus zio_checksum_compute() of the data to update the
1564  * gang header's blk_cksum as described above.
1565  *
1566  * The two-phase assemble/issue model solves the problem of partial failure --
1567  * what if you'd freed part of a gang block but then couldn't read the
1568  * gang header for another part?  Assembling the entire gang tree first
1569  * ensures that all the necessary gang header I/O has succeeded before
1570  * starting the actual work of free, claim, or write.  Once the gang tree
1571  * is assembled, free and claim are in-memory operations that cannot fail.
1572  *
1573  * In the event that a gang write fails, zio_dva_unallocate() walks the
1574  * gang tree to immediately free (i.e. insert back into the space map)
1575  * everything we've allocated.  This ensures that we don't get ENOSPC
1576  * errors during repeated suspend/resume cycles due to a flaky device.
1577  *
1578  * Gang rewrites only happen during sync-to-convergence.  If we can't assemble
1579  * the gang tree, we won't modify the block, so we can safely defer the free
1580  * (knowing that the block is still intact).  If we *can* assemble the gang
1581  * tree, then even if some of the rewrites fail, zio_dva_unallocate() will free
1582  * each constituent bp and we can allocate a new block on the next sync pass.
1583  *
1584  * In all cases, the gang tree allows complete recovery from partial failure.
1585  * ==========================================================================
1586  */

1588 static zio_t *
1589 zio_read_gang(zio_t *pio, blkptr_t *bp, zio_gang_node_t *gn, void *data)
1590 {
1591         if (gn != NULL)
1592                 return (pio);

1594         return (zio_read(pio, pio->io_spa, bp, data, BP_GET_PSIZE(bp),
1595             NULL, NULL, pio->io_priority, ZIO_GANG_CHILD_FLAGS(pio),
1596             &pio->io_bookmark));
1597 }

1599 zio_t *
```

```
1600 zio_rewrite_gang(zio_t *pio, blkptr_t *bp, zio_gang_node_t *gn, void *data)
1601 {
1602         zio_t *zio;

1604         if (gn != NULL) {
1605                 zio = zio_rewrite(pio, pio->io_spa, pio->io_txg, bp,
1606                     gn->gn_gbh, SPA_GANGBLOCKSIZE, NULL, NULL, pio->io_priority,
1607                     ZIO_GANG_CHILD_FLAGS(pio), &pio->io_bookmark);
1608                 /*
1609                  * As we rewrite each gang header, the pipeline will compute
1610                  * a new gang block header checksum for it; but no one will
1611                  * compute a new data checksum, so we do that here.  The one
1612                  * exception is the gang leader: the pipeline already computed
1613                  * its data checksum because that stage precedes gang assembly.
1614                  * (Presently, nothing actually uses interior data checksums;
1615                  * this is just good hygiene.)
1616                  */
1617                 if (gn != pio->io_gang_leader->io_gang_tree) {
1618                         zio_checksum_compute(zio, BP_GET_CHECKSUM(bp),
1619                             data, BP_GET_PSIZE(bp));
1620                 }
1621                 /*
1622                  * If we are here to damage data for testing purposes,
1623                  * leave the GBH alone so that we can detect the damage.
1624                  */
1625                 if (pio->io_gang_leader->io_flags & ZIO_FLAG_INDUCE_DAMAGE)
1626                         zio->io_pipeline &= ~ZIO_VDEV_IO_STAGES;
1627         } else {
1628                 zio = zio_rewrite(pio, pio->io_spa, pio->io_txg, bp,
1629                     data, BP_GET_PSIZE(bp), NULL, NULL, pio->io_priority,
1630                     ZIO_GANG_CHILD_FLAGS(pio), &pio->io_bookmark);
1631         }

1633         return (zio);
1634 }

1636 /* ARGSUSED */
1637 zio_t *
1638 zio_free_gang(zio_t *pio, blkptr_t *bp, zio_gang_node_t *gn, void *data)
1639 {
1640         return (zio_free_sync(pio, pio->io_spa, pio->io_txg, bp,
1641             ZIO_GANG_CHILD_FLAGS(pio)));
1642 }

1644 /* ARGSUSED */
1645 zio_t *
1646 zio_claim_gang(zio_t *pio, blkptr_t *bp, zio_gang_node_t *gn, void *data)
1647 {
1648         return (zio_claim(pio, pio->io_spa, pio->io_txg, bp,
1649             NULL, NULL, ZIO_GANG_CHILD_FLAGS(pio)));
1650 }

1652 static zio_gang_issue_func_t *zio_gang_issue_func[ZIO_TYPES] = {
1653         NULL,
1654         zio_read_gang,
1655         zio_rewrite_gang,
1656         zio_free_gang,
1657         zio_claim_gang,
1658         NULL
1659 };

1661 static void zio_gang_tree_assemble_done(zio_t *zio);

1663 static zio_gang_node_t *
1664 zio_gang_node_alloc(zio_gang_node_t **gnpp)
1665 {
```



```
1666         zio_gang_node_t *gn;

1668         ASSERT(*gnpp == NULL);

1670         gn = kmem_zalloc(sizeof (*gn), KM_SLEEP);
1671         gn->gn_gbh = zio_buf_alloc(SPA_GANGBLOCKSIZE);
1672         *gnpp = gn;

1674         return (gn);
1675 }

1677 static void
1678 zio_gang_node_free(zio_gang_node_t **gnpp)
1679 {
1680         zio_gang_node_t *gn = *gnpp;

1682         for (int g = 0; g < SPA_GBH_NBLKPTRS; g++)
1683                 ASSERT(gn->gn_child[g] == NULL);

1685         zio_buf_free(gn->gn_gbh, SPA_GANGBLOCKSIZE);
1686         kmem_free(gn, sizeof (*gn));
1687         *gnpp = NULL;
1688 }

1690 static void
1691 zio_gang_tree_free(zio_gang_node_t **gnpp)
1692 {
1693         zio_gang_node_t *gn = *gnpp;

1695         if (gn == NULL)
1696                 return;

1698         for (int g = 0; g < SPA_GBH_NBLKPTRS; g++)
1699                 zio_gang_tree_free(&gn->gn_child[g]);

1701         zio_gang_node_free(gnpp);
1702 }

1704 static void
1705 zio_gang_tree_assemble(zio_t *gio, blkptr_t *bp, zio_gang_node_t **gnpp)
1706 {
1707         zio_gang_node_t *gn = zio_gang_node_alloc(gnpp);

1709         ASSERT(gio->io_gang_leader == gio);
1710         ASSERT(BP_IS_GANG(bp));

1712         zio_nowait(zio_read(gio, gio->io_spa, bp, gn->gn_gbh,
1713             SPA_GANGBLOCKSIZE, zio_gang_tree_assemble_done, gn,
1714             gio->io_priority, ZIO_GANG_CHILD_FLAGS(gio), &gio->io_bookmark));
1715 }

1717 static void
1718 zio_gang_tree_assemble_done(zio_t *zio)
1719 {
1720         zio_t *gio = zio->io_gang_leader;
1721         zio_gang_node_t *gn = zio->io_private;
1722         blkptr_t *bp = zio->io_bp;

1724         ASSERT(gio == zio_unique_parent(zio));
1725         ASSERT(zio->io_child_count == 0);

1727         if (zio->io_error)
1728                 return;

1730         if (BP_SHOULD_BYTESWAP(bp))
1731                 byteswap_uint64_array(zio->io_data, zio->io_size);
```

```
1733         ASSERT(zio->io_data == gn->gn_gbh);
1734         ASSERT(zio->io_size == SPA_GANGBLOCKSIZE);
1735         ASSERT(gn->gn_gbh->zg_tail.zec_magic == ZEC_MAGIC);
1736
1737         for (int g = 0; g < SPA_GBH_NBLKPTRS; g++) {
1738                 blkptr_t *gbp = &gn->gn_gbh->zg_blkptr[g];
1739                 if (!BP_IS_GANG(gbp))
1740                         continue;
1741                 zio_gang_tree_assemble(gio, gbp, &gn->gn_child[g]);
1742         }
1743 }
1744
1745 static void
1746 zio_gang_tree_issue(zio_t *pio, zio_gang_node_t *gn, blkptr_t *bp, void *data)
1747 {
1748         zio_t *gio = pio->io_gang_leader;
1749         zio_t *zio;
1750
1751         ASSERT(BP_IS_GANG(bp) == !!gn);
1752         ASSERT(BP_GET_CHECKSUM(bp) == BP_GET_CHECKSUM(gio->io_bp));
1753         ASSERT(BP_GET_LSIZE(bp) == BP_GET_PSIZE(bp) || gn == gio->io_gang_tree);
1754
1755         /*
1756          * If you're a gang header, your data is in gn->gn_gbh.
1757          * If you're a gang member, your data is in 'data' and gn == NULL.
1758          */
1759         zio = zio_gang_issue_func[gio->io_type](pio, bp, gn, data);
1760
1761         if (gn != NULL) {
1762                 ASSERT(gn->gn_gbh->zg_tail.zec_magic == ZEC_MAGIC);
1763
1764                 for (int g = 0; g < SPA_GBH_NBLKPTRS; g++) {
1765                         blkptr_t *gbp = &gn->gn_gbh->zg_blkptr[g];
1766                         if (BP_IS_HOLE(gbp))
1767                                 continue;
1768                         zio_gang_tree_issue(zio, gn->gn_child[g], gbp, data);
1769                         data = (char *)data + BP_GET_PSIZE(gbp);
1770                 }
1771         }
1772
1773         if (gn == gio->io_gang_tree)
1774                 ASSERT3P((char *)gio->io_data + gio->io_size, ==, data);
1775
1776         if (zio != pio)
1777                 zio_nowait(zio);
1778 }
1779
1780 static int
1781 zio_gang_assemble(zio_t *zio)
1782 {
1783         blkptr_t *bp = zio->io_bp;
1784
1785         ASSERT(BP_IS_GANG(bp) && zio->io_gang_leader == NULL);
1786         ASSERT(zio->io_child_type > ZIO_CHILD_GANG);
1787
1788         zio->io_gang_leader = zio;
1789
1790         zio_gang_tree_assemble(zio, bp, &zio->io_gang_tree);
1791
1792         return (ZIO_PIPELINE_CONTINUE);
1793 }
1794
1795 static int
1796 zio_gang_issue(zio_t *zio)
1797 {
```



```
1798         blkptr_t *bp = zio->io_bp;
1799
1800         if (zio_wait_for_children(zio, ZIO_CHILD_GANG, ZIO_WAIT_DONE))
1801                 return (ZIO_PIPELINE_STOP);
1802
1803         ASSERT(BP_IS_GANG(bp) && zio->io_gang_leader == zio);
1804         ASSERT(zio->io_child_type > ZIO_CHILD_GANG);
1805
1806         if (zio->io_child_error[ZIO_CHILD_GANG] == 0)
1807                 zio_gang_tree_issue(zio, zio->io_gang_tree, bp, zio->io_data);
1808         else
1809                 zio_gang_tree_free(&zio->io_gang_tree);
1810
1811         zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;
1812
1813         return (ZIO_PIPELINE_CONTINUE);
1814 }
1815
1816 static void
1817 zio_write_gang_member_ready(zio_t *zio)
1818 {
1819         zio_t *pio = zio_unique_parent(zio);
1820         zio_t *gio = zio->io_gang_leader;
1821         dva_t *cdva = zio->io_bp->blk_dva;
1822         dva_t *pdva = pio->io_bp->blk_dva;
1823         uint64_t asize;
1824
1825         if (BP_IS_HOLE(zio->io_bp))
1826                 return;
1827
1828         ASSERT(BP_IS_HOLE(&zio->io_bp_orig));
1829
1830         ASSERT(zio->io_child_type == ZIO_CHILD_GANG);
1831         ASSERT3U(zio->io_prop.zp_copies, ==, gio->io_prop.zp_copies);
1832         ASSERT3U(zio->io_prop.zp_copies, <=, BP_GET_NDVAS(zio->io_bp));
1833         ASSERT3U(pio->io_prop.zp_copies, <=, BP_GET_NDVAS(pio->io_bp));
1834         ASSERT3U(BP_GET_NDVAS(zio->io_bp), <=, BP_GET_NDVAS(pio->io_bp));
1835
1836         mutex_enter(&pio->io_lock);
1837         for (int d = 0; d < BP_GET_NDVAS(zio->io_bp); d++) {
1838                 ASSERT(DVA_GET_GANG(&pdva[d]));
1839                 asize = DVA_GET_ASIZE(&pdva[d]);
1840                 asize += DVA_GET_ASIZE(&cdva[d]);
1841                 DVA_SET_ASIZE(&pdva[d], asize);
1842         }
1843         mutex_exit(&pio->io_lock);
1844 }
1845
1846 static int
1847 zio_write_gang_block(zio_t *pio)
1848 {
1849         spa_t *spa = pio->io_spa;
1850         blkptr_t *bp = pio->io_bp;
1851         zio_t *gio = pio->io_gang_leader;
1852         zio_t *zio;
1853         zio_gang_node_t *gn, **gnpp;
1854         zio_gbh_phys_t *gbh;
1855         uint64_t txg = pio->io_txg;
1856         uint64_t resid = pio->io_size;
1857         uint64_t lsize;
1858         int copies = gio->io_prop.zp_copies;
1859         int gbh_copies = MIN(copies + 1, spa_max_replication(spa));
1860         zio_prop_t zp;
1861         int error;
1862
1863         error = metaslab_alloc(spa, spa_normal_class(spa), SPA_GANGBLOCKSIZE,
```

```
1864             bp, gbh_copies, txg, pio == gio ? NULL : gio->io_bp,
1865             METASLAB_HINTBP_FAVOR | METASLAB_GANG_HEADER);
1866         if (error) {
1867                 pio->io_error = error;
1868                 return (ZIO_PIPELINE_CONTINUE);
1869         }

1871         if (pio == gio) {
1872                 gnpp = &gio->io_gang_tree;
1873         } else {
1874                 gnpp = pio->io_private;
1875                 ASSERT(pio->io_ready == zio_write_gang_member_ready);
1876         }

1878         gn = zio_gang_node_alloc(gnpp);
1879         gbh = gn->gn_gbh;
1880         bzero(gbh, SPA_GANGBLOCKSIZE);

1882         /*
1883          * Create the gang header.
1884          */
1885         zio = zio_rewrite(pio, spa, txg, bp, gbh, SPA_GANGBLOCKSIZE, NULL, NULL,
1886             pio->io_priority, ZIO_GANG_CHILD_FLAGS(pio), &pio->io_bookmark);

1888         /*
1889          * Create and nowait the gang children.
1890          */
1891         for (int g = 0; resid != 0; resid -= lsize, g++) {
1892                 lsize = P2ROUNDUP(resid / (SPA_GBH_NBLKPTRS - g),
1893                     SPA_MINBLOCKSIZE);
1894                 ASSERT(lsize >= SPA_MINBLOCKSIZE && lsize <= resid);

1896                 zp.zp_checksum = gio->io_prop.zp_checksum;
1897                 zp.zp_compress = ZIO_COMPRESS_OFF;
1898                 zp.zp_type = DMU_OT_NONE;
1899                 zp.zp_level = 0;
1900                 zp.zp_copies = gio->io_prop.zp_copies;
1901                 zp.zp_dedup = B_FALSE;
1902                 zp.zp_dedup_verify = B_FALSE;
1903                 zp.zp_zero_write = B_FALSE;
1904 #endif /* ! codereview */
1905                 zp.zp_nopwrite = B_FALSE;

1907                 zio_nowait(zio_write(zio, spa, txg, &gbh->zg_blkptr[g],
1908                     (char *)pio->io_data + (pio->io_size - resid), lsize, &zp,
1909                     zio_write_gang_member_ready, NULL, NULL, &gn->gn_child[g],
1910                     pio->io_priority, ZIO_GANG_CHILD_FLAGS(pio),
1911                     &pio->io_bookmark));
1912         }

1914         /*
1915          * Set pio's pipeline to just wait for zio to finish.
1916          */
1917         pio->io_pipeline = ZIO_INTERLOCK_PIPELINE;

1919         zio_nowait(zio);

1921         return (ZIO_PIPELINE_CONTINUE);
1922 }

1924 /*
1925  * The zio_nop_write stage in the pipeline determines if allocating
1926  * a new bp is necessary.  By leveraging a cryptographically secure checksum,
1927  * such as SHA256, we can compare the checksums of the new data and the old
1928  * to determine if allocating a new block is required.  The nopwrite
1929  * feature can handle writes in either syncing or open context (i.e. zil
```



```
1930  * writes) and as a result is mutually exclusive with dedup.
1931  */
1932 static int
1933 zio_nop_write(zio_t *zio)
1934 {
1935         blkptr_t *bp = zio->io_bp;
1936         blkptr_t *bp_orig = &zio->io_bp_orig;
1937         zio_prop_t *zp = &zio->io_prop;

1939         ASSERT(BP_GET_LEVEL(bp) == 0);
1940         ASSERT(!(zio->io_flags & ZIO_FLAG_IO_REWRITE));
1941         ASSERT(zp->zp_nopwrite);
1942         ASSERT(!zp->zp_dedup);
1943         ASSERT(zio->io_bp_override == NULL);
1944         ASSERT(IO_IS_ALLOCATING(zio));

1946         /*
1947          * Check to see if the original bp and the new bp have matching
1948          * characteristics (i.e. same checksum, compression algorithms, etc).
1949          * If they don't then just continue with the pipeline which will
1950          * allocate a new bp.
1951          */
1952         if (BP_IS_HOLE(bp_orig) ||
1953             !zio_checksum_table[BP_GET_CHECKSUM(bp)].ci_dedup ||
1954             BP_GET_CHECKSUM(bp) != BP_GET_CHECKSUM(bp_orig) ||
1955             BP_GET_COMPRESS(bp) != BP_GET_COMPRESS(bp_orig) ||
1956             BP_GET_DEDUP(bp) != BP_GET_DEDUP(bp_orig) ||
1957             zp->zp_copies != BP_GET_NDVAS(bp_orig))
1958                 return (ZIO_PIPELINE_CONTINUE);

1960         /*
1961          * If the checksums match then reset the pipeline so that we
1962          * avoid allocating a new bp and issuing any I/O.
1963          */
1964         if (ZIO_CHECKSUM_EQUAL(bp->blk_cksum, bp_orig->blk_cksum)) {
1965                 ASSERT(zio_checksum_table[zp->zp_checksum].ci_dedup);
1966                 ASSERT3U(BP_GET_PSIZE(bp), ==, BP_GET_PSIZE(bp_orig));
1967                 ASSERT3U(BP_GET_LSIZE(bp), ==, BP_GET_LSIZE(bp_orig));
1968                 ASSERT(zp->zp_compress != ZIO_COMPRESS_OFF);
1969                 ASSERT(bcmp(&bp->blk_prop, &bp_orig->blk_prop,
1970                     sizeof (uint64_t)) == 0);

1972                 *bp = *bp_orig;
1973                 zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;
1974                 zio->io_flags |= ZIO_FLAG_NOPWRITE;
1975         }

1977         return (ZIO_PIPELINE_CONTINUE);
1978 }

1980 /*
1981  * ==========================================================================
1982  * Dedup
1983  * ==========================================================================
1984  */
1985 static void
1986 zio_ddt_child_read_done(zio_t *zio)
1987 {
1988         blkptr_t *bp = zio->io_bp;
1989         ddt_entry_t *dde = zio->io_private;
1990         ddt_phys_t *ddp;
1991         zio_t *pio = zio_unique_parent(zio);

1993         mutex_enter(&pio->io_lock);
1994         ddp = ddt_phys_select(dde, bp);
1995         if (zio->io_error == 0)
```

```
1996                ddt_phys_clear(ddp);    /* this ddp doesn't need repair */
1997        if (zio->io_error == 0 && dde->dde_repair_data == NULL)
1998                dde->dde_repair_data = zio->io_data;
1999        else
2000                zio_buf_free(zio->io_data, zio->io_size);
2001        mutex_exit(&pio->io_lock);
2002 }

2004 static int
2005 zio_ddt_read_start(zio_t *zio)
2006 {
2007        blkptr_t *bp = zio->io_bp;

2009        ASSERT(BP_GET_DEDUP(bp));
2010        ASSERT(BP_GET_PSIZE(bp) == zio->io_size);
2011        ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);

2013        if (zio->io_child_error[ZIO_CHILD_DDT]) {
2014                ddt_t *ddt = ddt_select(zio->io_spa, bp);
2015                ddt_entry_t *dde = ddt_repair_start(ddt, bp);
2016                ddt_phys_t *ddp = dde->dde_phys;
2017                ddt_phys_t *ddp_self = ddt_phys_select(dde, bp);
2018                blkptr_t blk;

2020                ASSERT(zio->io_vsd == NULL);
2021                zio->io_vsd = dde;

2023                if (ddp_self == NULL)
2024                        return (ZIO_PIPELINE_CONTINUE);

2026                for (int p = 0; p < DDT_PHYS_TYPES; p++, ddp++) {
2027                        if (ddp->ddp_phys_birth == 0 || ddp == ddp_self)
2028                                continue;
2029                        ddt_bp_create(ddt->ddt_checksum, &dde->dde_key, ddp,
2030                            &blk);
2031                        zio_nowait(zio_read(zio, zio->io_spa, &blk,
2032                            zio_buf_alloc(zio->io_size), zio->io_size,
2033                            zio_ddt_child_read_done, dde, zio->io_priority,
2034                            ZIO_DDT_CHILD_FLAGS(zio) | ZIO_FLAG_DONT_PROPAGATE,
2035                            &zio->io_bookmark));
2036                }
2037                return (ZIO_PIPELINE_CONTINUE);
2038        }

2040        zio_nowait(zio_read(zio, zio->io_spa, bp,
2041            zio->io_data, zio->io_size, NULL, NULL, zio->io_priority,
2042            ZIO_DDT_CHILD_FLAGS(zio), &zio->io_bookmark));

2044        return (ZIO_PIPELINE_CONTINUE);
2045 }

2047 static int
2048 zio_ddt_read_done(zio_t *zio)
2049 {
2050        blkptr_t *bp = zio->io_bp;

2052        if (zio_wait_for_children(zio, ZIO_CHILD_DDT, ZIO_WAIT_DONE))
2053                return (ZIO_PIPELINE_STOP);

2055        ASSERT(BP_GET_DEDUP(bp));
2056        ASSERT(BP_GET_PSIZE(bp) == zio->io_size);
2057        ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);

2059        if (zio->io_child_error[ZIO_CHILD_DDT]) {
2060                ddt_t *ddt = ddt_select(zio->io_spa, bp);
2061                ddt_entry_t *dde = zio->io_vsd;
```



```
2062                if (ddt == NULL) {
2063                        ASSERT(spa_load_state(zio->io_spa) != SPA_LOAD_NONE);
2064                        return (ZIO_PIPELINE_CONTINUE);
2065                }
2066                if (dde == NULL) {
2067                        zio->io_stage = ZIO_STAGE_DDT_READ_START >> 1;
2068                        zio_taskq_dispatch(zio, ZIO_TASKQ_ISSUE, B_FALSE);
2069                        return (ZIO_PIPELINE_STOP);
2070                }
2071                if (dde->dde_repair_data != NULL) {
2072                        bcopy(dde->dde_repair_data, zio->io_data, zio->io_size);
2073                        zio->io_child_error[ZIO_CHILD_DDT] = 0;
2074                }
2075                ddt_repair_done(ddt, dde);
2076                zio->io_vsd = NULL;
2077        }

2079        ASSERT(zio->io_vsd == NULL);

2081        return (ZIO_PIPELINE_CONTINUE);
2082 }

2084 static boolean_t
2085 zio_ddt_collision(zio_t *zio, ddt_t *ddt, ddt_entry_t *dde)
2086 {
2087        spa_t *spa = zio->io_spa;

2089        /*
2090         * Note: we compare the original data, not the transformed data,
2091         * because when zio->io_bp is an override bp, we will not have
2092         * pushed the I/O transforms.  That's an important optimization
2093         * because otherwise we'd compress/encrypt all dmu_sync() data twice.
2094         */
2095        for (int p = DDT_PHYS_SINGLE; p <= DDT_PHYS_TRIPLE; p++) {
2096                zio_t *lio = dde->dde_lead_zio[p];

2098                if (lio != NULL) {
2099                        return (lio->io_orig_size != zio->io_orig_size ||
2100                            bcmp(zio->io_orig_data, lio->io_orig_data,
2101                            zio->io_orig_size) != 0);
2102                }
2103        }

2105        for (int p = DDT_PHYS_SINGLE; p <= DDT_PHYS_TRIPLE; p++) {
2106                ddt_phys_t *ddp = &dde->dde_phys[p];

2108                if (ddp->ddp_phys_birth != 0) {
2109                        arc_buf_t *abuf = NULL;
2110                        uint32_t aflags = ARC_WAIT;
2111                        blkptr_t blk = *zio->io_bp;
2112                        int error;

2114                        ddt_bp_fill(ddp, &blk, ddp->ddp_phys_birth);

2116                        ddt_exit(ddt);

2118                        error = arc_read(NULL, spa, &blk,
2119                            arc_getbuf_func, &abuf, ZIO_PRIORITY_SYNC_READ,
2120                            ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE,
2121                            &aflags, &zio->io_bookmark);

2123                        if (error == 0) {
2124                                if (arc_buf_size(abuf) != zio->io_orig_size ||
2125                                    bcmp(abuf->b_data, zio->io_orig_data,
2126                                    zio->io_orig_size) != 0)
2127                                        error = SET_ERROR(EEXIST);
```

```
2128                                VERIFY(arc_buf_remove_ref(abuf, &abuf));
2129                        }
2131                        ddt_enter(ddt);
2132                        return (error != 0);
2133                }
2134        }
2136        return (B_FALSE);
2137 }
2139 static void
2140 zio_ddt_child_write_ready(zio_t *zio)
2141 {
2142        int p = zio->io_prop.zp_copies;
2143        ddt_t *ddt = ddt_select(zio->io_spa, zio->io_bp);
2144        ddt_entry_t *dde = zio->io_private;
2145        ddt_phys_t *ddp = &dde->dde_phys[p];
2146        zio_t *pio;
2148        if (zio->io_error)
2149                return;
2151        ddt_enter(ddt);
2153        ASSERT(dde->dde_lead_zio[p] == zio);
2155        ddt_phys_fill(ddp, zio->io_bp);
2157        while ((pio = zio_walk_parents(zio)) != NULL)
2158                ddt_bp_fill(ddp, pio->io_bp, zio->io_txg);
2160        ddt_exit(ddt);
2161 }
2163 static void
2164 zio_ddt_child_write_done(zio_t *zio)
2165 {
2166        int p = zio->io_prop.zp_copies;
2167        ddt_t *ddt = ddt_select(zio->io_spa, zio->io_bp);
2168        ddt_entry_t *dde = zio->io_private;
2169        ddt_phys_t *ddp = &dde->dde_phys[p];
2171        ddt_enter(ddt);
2173        ASSERT(ddp->ddp_refcnt == 0);
2174        ASSERT(dde->dde_lead_zio[p] == zio);
2175        dde->dde_lead_zio[p] = NULL;
2177        if (zio->io_error == 0) {
2178                while (zio_walk_parents(zio) != NULL)
2179                        ddt_phys_addref(ddp);
2180        } else {
2181                ddt_phys_clear(ddp);
2182        }
2184        ddt_exit(ddt);
2185 }
2187 static void
2188 zio_ddt_ditto_write_done(zio_t *zio)
2189 {
2190        int p = DDT_PHYS_DITTO;
2191        zio_prop_t *zp = &zio->io_prop;
2192        blkptr_t *bp = zio->io_bp;
2193        ddt_t *ddt = ddt_select(zio->io_spa, bp);
```



```
2194        ddt_entry_t *dde = zio->io_private;
2195        ddt_phys_t *ddp = &dde->dde_phys[p];
2196        ddt_key_t *ddk = &dde->dde_key;
2198        ddt_enter(ddt);
2200        ASSERT(ddp->ddp_refcnt == 0);
2201        ASSERT(dde->dde_lead_zio[p] == zio);
2202        dde->dde_lead_zio[p] = NULL;
2204        if (zio->io_error == 0) {
2205                ASSERT(ZIO_CHECKSUM_EQUAL(bp->blk_cksum, ddk->ddk_cksum));
2206                ASSERT(zp->zp_copies < SPA_DVAS_PER_BP);
2207                ASSERT(zp->zp_copies == BP_GET_NDVAS(bp) - BP_IS_GANG(bp));
2208                if (ddp->ddp_phys_birth != 0)
2209                        ddt_phys_free(ddt, ddk, ddp, zio->io_txg);
2210                ddt_phys_fill(ddp, bp);
2211        }
2213        ddt_exit(ddt);
2214 }
2216 static int
2217 zio_ddt_write(zio_t *zio)
2218 {
2219        spa_t *spa = zio->io_spa;
2220        blkptr_t *bp = zio->io_bp;
2221        uint64_t txg = zio->io_txg;
2222        zio_prop_t *zp = &zio->io_prop;
2223        int p = zp->zp_copies;
2224        int ditto_copies;
2225        zio_t *cio = NULL;
2226        zio_t *dio = NULL;
2227        ddt_t *ddt = ddt_select(spa, bp);
2228        ddt_entry_t *dde;
2229        ddt_phys_t *ddp;
2231        ASSERT(BP_GET_DEDUP(bp));
2232        ASSERT(BP_GET_CHECKSUM(bp) == zp->zp_checksum);
2233        ASSERT(BP_IS_HOLE(bp) || zio->io_bp_override);
2235        ddt_enter(ddt);
2236        dde = ddt_lookup(ddt, bp, B_TRUE);
2237        ddp = &dde->dde_phys[p];
2239        if (zp->zp_dedup_verify && zio_ddt_collision(zio, ddt, dde)) {
2240                /*
2241                 * If we're using a weak checksum, upgrade to a strong checksum
2242                 * and try again.  If we're already using a strong checksum,
2243                 * we can't resolve it, so just convert to an ordinary write.
2244                 * (And automatically e-mail a paper to Nature?)
2245                 */
2246                if (!zio_checksum_table[zp->zp_checksum].ci_dedup) {
2247                        zp->zp_checksum = spa_dedup_checksum(spa);
2248                        zio_pop_transforms(zio);
2249                        zio->io_stage = ZIO_STAGE_OPEN;
2250                        BP_ZERO(bp);
2251                } else {
2252                        zp->zp_dedup = B_FALSE;
2253                }
2254                zio->io_pipeline = ZIO_WRITE_PIPELINE;
2255                ddt_exit(ddt);
2256                return (ZIO_PIPELINE_CONTINUE);
2257        }
2259        ditto_copies = ddt_ditto_copies_needed(ddt, dde, ddp);
```

```
2260         ASSERT(ditto_copies < SPA_DVAS_PER_BP);

2262         if (ditto_copies > ddt_ditto_copies_present(dde) &&
2263             dde->dde_lead_zio[DDT_PHYS_DITTO] == NULL) {
2264                 zio_prop_t czp = *zp;

2266                 czp.zp_copies = ditto_copies;

2268                 /*
2269                  * If we arrived here with an override bp, we won't have run
2270                  * the transform stack, so we won't have the data we need to
2271                  * generate a child i/o.  So, toss the override bp and restart.
2272                  * This is safe, because using the override bp is just an
2273                  * optimization; and it's rare, so the cost doesn't matter.
2274                  */
2275                 if (zio->io_bp_override) {
2276                         zio_pop_transforms(zio);
2277                         zio->io_stage = ZIO_STAGE_OPEN;
2278                         zio->io_pipeline = ZIO_WRITE_PIPELINE;
2279                         zio->io_bp_override = NULL;
2280                         BP_ZERO(bp);
2281                         ddt_exit(ddt);
2282                         return (ZIO_PIPELINE_CONTINUE);
2283                 }

2285                 dio = zio_write(zio, spa, txg, bp, zio->io_orig_data,
2286                     zio->io_orig_size, &czp, NULL, NULL,
2287                     zio_ddt_ditto_write_done, dde, zio->io_priority,
2288                     ZIO_DDT_CHILD_FLAGS(zio), &zio->io_bookmark);

2290                 zio_push_transform(dio, zio->io_data, zio->io_size, 0, NULL);
2291                 dde->dde_lead_zio[DDT_PHYS_DITTO] = dio;
2292         }

2294         if (ddp->ddp_phys_birth != 0 || dde->dde_lead_zio[p] != NULL) {
2295                 if (ddp->ddp_phys_birth != 0)
2296                         ddt_bp_fill(ddp, bp, txg);
2297                 if (dde->dde_lead_zio[p] != NULL)
2298                         zio_add_child(zio, dde->dde_lead_zio[p]);
2299                 else
2300                         ddt_phys_addref(ddp);
2301         } else if (zio->io_bp_override) {
2302                 ASSERT(bp->blk_birth == txg);
2303                 ASSERT(BP_EQUAL(bp, zio->io_bp_override));
2304                 ddt_phys_fill(ddp, bp);
2305                 ddt_phys_addref(ddp);
2306         } else {
2307                 cio = zio_write(zio, spa, txg, bp, zio->io_orig_data,
2308                     zio->io_orig_size, zp, zio_ddt_child_write_ready, NULL,
2309                     zio_ddt_child_write_done, dde, zio->io_priority,
2310                     ZIO_DDT_CHILD_FLAGS(zio), &zio->io_bookmark);

2312                 zio_push_transform(cio, zio->io_data, zio->io_size, 0, NULL);
2313                 dde->dde_lead_zio[p] = cio;
2314         }

2316         ddt_exit(ddt);

2318         if (cio)
2319                 zio_nowait(cio);
2320         if (dio)
2321                 zio_nowait(dio);

2323         return (ZIO_PIPELINE_CONTINUE);
2324 }
```



```
2326 ddt_entry_t *freedde; /* for debugging */

2328 static int
2329 zio_ddt_free(zio_t *zio)
2330 {
2331         spa_t *spa = zio->io_spa;
2332         blkptr_t *bp = zio->io_bp;
2333         ddt_t *ddt = ddt_select(spa, bp);
2334         ddt_entry_t *dde;
2335         ddt_phys_t *ddp;

2337         ASSERT(BP_GET_DEDUP(bp));
2338         ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);

2340         ddt_enter(ddt);
2341         freedde = dde = ddt_lookup(ddt, bp, B_TRUE);
2342         ddp = ddt_phys_select(dde, bp);
2343         ddt_phys_decref(ddp);
2344         ddt_exit(ddt);

2346         return (ZIO_PIPELINE_CONTINUE);
2347 }

2349 /*
2350  * ==========================================================================
2351  * Allocate and free blocks
2352  * ==========================================================================
2353  */
2354 static int
2355 zio_dva_allocate(zio_t *zio)
2356 {
2357         spa_t *spa = zio->io_spa;
2358         metaslab_class_t *mc = spa_normal_class(spa);
2359         blkptr_t *bp = zio->io_bp;
2360         int error;
2361         int flags = 0;

2363         if (zio->io_gang_leader == NULL) {
2364                 ASSERT(zio->io_child_type > ZIO_CHILD_GANG);
2365                 zio->io_gang_leader = zio;
2366         }

2368         ASSERT(BP_IS_HOLE(bp));
2369         ASSERT0(BP_GET_NDVAS(bp));
2370         ASSERT3U(zio->io_prop.zp_copies, >, 0);
2371         ASSERT3U(zio->io_prop.zp_copies, <=, spa_max_replication(spa));
2372         ASSERT3U(zio->io_size, ==, BP_GET_PSIZE(bp));

2374         /*
2375          * The dump device does not support gang blocks so allocation on
2376          * behalf of the dump device (i.e. ZIO_FLAG_NODATA) must avoid
2377          * the "fast" gang feature.
2378          */
2379         flags |= (zio->io_flags & ZIO_FLAG_NODATA) ? METASLAB_GANG_AVOID : 0;
2380         flags |= (zio->io_flags & ZIO_FLAG_GANG_CHILD) ?
2381             METASLAB_GANG_CHILD : 0;
2382         error = metaslab_alloc(spa, mc, zio->io_size, bp,
2383             zio->io_prop.zp_copies, zio->io_txg, NULL, flags);

2385         if (error) {
2386                 spa_dbgmsg(spa, "%s: metaslab allocation failure: zio %p, "
2387                     "size %llu, error %d", spa_name(spa), zio, zio->io_size,
2388                     error);
2389                 if (error == ENOSPC && zio->io_size > SPA_MINBLOCKSIZE)
2390                         return (zio_write_gang_block(zio));
2391                 zio->io_error = error;
```

```
2392         }

2394         return (ZIO_PIPELINE_CONTINUE);
2395 }

2397 static int
2398 zio_dva_free(zio_t *zio)
2399 {
2400         metaslab_free(zio->io_spa, zio->io_bp, zio->io_txg, B_FALSE);

2402         return (ZIO_PIPELINE_CONTINUE);
2403 }

2405 static int
2406 zio_dva_claim(zio_t *zio)
2407 {
2408         int error;

2410         error = metaslab_claim(zio->io_spa, zio->io_bp, zio->io_txg);
2411         if (error)
2412                 zio->io_error = error;

2414         return (ZIO_PIPELINE_CONTINUE);
2415 }

2417 /*
2418  * Undo an allocation.  This is used by zio_done() when an I/O fails
2419  * and we want to give back the block we just allocated.
2420  * This handles both normal blocks and gang blocks.
2421  */
2422 static void
2423 zio_dva_unallocate(zio_t *zio, zio_gang_node_t *gn, blkptr_t *bp)
2424 {
2425         ASSERT(bp->blk_birth == zio->io_txg || BP_IS_HOLE(bp));
2426         ASSERT(zio->io_bp_override == NULL);

2428         if (!BP_IS_HOLE(bp))
2429                 metaslab_free(zio->io_spa, bp, bp->blk_birth, B_TRUE);

2431         if (gn != NULL) {
2432                 for (int g = 0; g < SPA_GBH_NBLKPTRS; g++) {
2433                         zio_dva_unallocate(zio, gn->gn_child[g],
2434                             &gn->gn_gbh->zg_blkptr[g]);
2435                 }
2436         }
2437 }

2439 /*
2440  * Try to allocate an intent log block.  Return 0 on success, errno on failure.
2441  */
2442 int
2443 zio_alloc_zil(spa_t *spa, uint64_t txg, blkptr_t *new_bp, blkptr_t *old_bp,
2444     uint64_t size, boolean_t use_slog)
2445 {
2446         int error = 1;

2448         ASSERT(txg > spa_syncing_txg(spa));

2450         /*
2451          * ZIL blocks are always contiguous (i.e. not gang blocks) so we
2452          * set the METASLAB_GANG_AVOID flag so that they don't "fast gang"
2453          * when allocating them.
2454          */
2455         if (use_slog) {
2456                 error = metaslab_alloc(spa, spa_log_class(spa), size,
2457                     new_bp, 1, txg, old_bp,
```



```
2458                     METASLAB_HINTBP_AVOID | METASLAB_GANG_AVOID);
2459         }

2461         if (error) {
2462                 error = metaslab_alloc(spa, spa_normal_class(spa), size,
2463                     new_bp, 1, txg, old_bp,
2464                     METASLAB_HINTBP_AVOID);
2465         }

2467         if (error == 0) {
2468                 BP_SET_LSIZE(new_bp, size);
2469                 BP_SET_PSIZE(new_bp, size);
2470                 BP_SET_COMPRESS(new_bp, ZIO_COMPRESS_OFF);
2471                 BP_SET_CHECKSUM(new_bp,
2472                     spa_version(spa) >= SPA_VERSION_SLIM_ZIL
2473                     ? ZIO_CHECKSUM_ZILOG2 : ZIO_CHECKSUM_ZILOG);
2474                 BP_SET_TYPE(new_bp, DMU_OT_INTENT_LOG);
2475                 BP_SET_LEVEL(new_bp, 0);
2476                 BP_SET_DEDUP(new_bp, 0);
2477                 BP_SET_BYTEORDER(new_bp, ZFS_HOST_BYTEORDER);
2478         }

2480         return (error);
2481 }

2483 /*
2484  * Free an intent log block.
2485  */
2486 void
2487 zio_free_zil(spa_t *spa, uint64_t txg, blkptr_t *bp)
2488 {
2489         ASSERT(BP_GET_TYPE(bp) == DMU_OT_INTENT_LOG);
2490         ASSERT(!BP_IS_GANG(bp));

2492         zio_free(spa, txg, bp);
2493 }

2495 /*
2496  * ==========================================================================
2497  * Read and write to physical devices
2498  * ==========================================================================
2499  */
2500 static int
2501 zio_vdev_io_start(zio_t *zio)
2502 {
2503         vdev_t *vd = zio->io_vd;
2504         uint64_t align;
2505         spa_t *spa = zio->io_spa;

2507         ASSERT(zio->io_error == 0);
2508         ASSERT(zio->io_child_error[ZIO_CHILD_VDEV] == 0);

2510         if (vd == NULL) {
2511                 if (!(zio->io_flags & ZIO_FLAG_CONFIG_WRITER))
2512                         spa_config_enter(spa, SCL_ZIO, zio, RW_READER);

2514                 /*
2515                  * The mirror_ops handle multiple DVAs in a single BP.
2516                  */
2517                 return (vdev_mirror_ops.vdev_op_io_start(zio));
2518         }

2520         /*
2521          * We keep track of time-sensitive I/Os so that the scan thread
2522          * can quickly react to certain workloads.  In particular, we care
2523          * about non-scrubbing, top-level reads and writes with the following
```

```
2524          * characteristics:
2525          *      - synchronous writes of user data to non-slog devices
2526          *      - any reads of user data
2527          * When these conditions are met, adjust the timestamp of spa_last_io
2528          * which allows the scan thread to adjust its workload accordingly.
2529          */
2530         if (!(zio->io_flags & ZIO_FLAG_SCAN_THREAD) && zio->io_bp != NULL &&
2531             vd == vd->vdev_top && !vd->vdev_islog &&
2532             zio->io_bookmark.zb_objset != DMU_META_OBJSET &&
2533             zio->io_txg != spa_syncing_txg(spa)) {
2534                 uint64_t old = spa->spa_last_io;
2535                 uint64_t new = ddi_get_lbolt64();
2536                 if (old != new)
2537                         (void) atomic_cas_64(&spa->spa_last_io, old, new);
2538         }

2540         align = 1ULL << vd->vdev_top->vdev_ashift;

2542         if (!(zio->io_flags & ZIO_FLAG_PHYSICAL) &&
2543             P2PHASE(zio->io_size, align) != 0) {
2544                 /* Transform logical writes to be a full physical block size. */
2545                 uint64_t asize = P2ROUNDUP(zio->io_size, align);
2546                 char *abuf = zio_buf_alloc(asize);
2547                 ASSERT(vd == vd->vdev_top);
2548                 if (zio->io_type == ZIO_TYPE_WRITE) {
2549                         bcopy(zio->io_data, abuf, zio->io_size);
2550                         bzero(abuf + zio->io_size, asize - zio->io_size);
2551                 }
2552                 zio_push_transform(zio, abuf, asize, asize, zio_subblock);
2553         }

2555         /*
2556          * If this is not a physical io, make sure that it is properly aligned
2557          * before proceeding.
2558          */
2559         if (!(zio->io_flags & ZIO_FLAG_PHYSICAL)) {
2560                 ASSERT0(P2PHASE(zio->io_offset, align));
2561                 ASSERT0(P2PHASE(zio->io_size, align));
2562         } else {
2563                 /*
2564                  * For physical writes, we allow 512b aligned writes and assume
2565                  * the device will perform a read-modify-write as necessary.
2566                  */
2567                 ASSERT0(P2PHASE(zio->io_offset, SPA_MINBLOCKSIZE));
2568                 ASSERT0(P2PHASE(zio->io_size, SPA_MINBLOCKSIZE));
2569         }

2571         VERIFY(zio->io_type != ZIO_TYPE_WRITE || spa_writeable(spa));

2573         /*
2574          * If this is a repair I/O, and there's no self-healing involved --
2575          * that is, we're just resilvering what we expect to resilver --
2576          * then don't do the I/O unless zio's txg is actually in vd's DTL.
2577          * This prevents spurious resilvering with nested replication.
2578          * For example, given a mirror of mirrors, (A+B)+(C+D), if only
2579          * A is out of date, we'll read from C+D, then use the data to
2580          * resilver A+B -- but we don't actually want to resilver B, just A.
2581          * The top-level mirror has no way to know this, so instead we just
2582          * discard unnecessary repairs as we work our way down the vdev tree.
2583          * The same logic applies to any form of nested replication:
2584          * ditto + mirror, RAID-Z + replacing, etc.  This covers them all.
2585          */
2586         if ((zio->io_flags & ZIO_FLAG_IO_REPAIR) &&
2587             !(zio->io_flags & ZIO_FLAG_SELF_HEAL) &&
2588             zio->io_txg != 0 && /* not a delegated i/o */
2589             !vdev_dtl_contains(vd, DTL_PARTIAL, zio->io_txg, 1)) {
```



```
2590                 ASSERT(zio->io_type == ZIO_TYPE_WRITE);
2591                 zio_vdev_io_bypass(zio);
2592                 return (ZIO_PIPELINE_CONTINUE);
2593         }

2595         if (vd->vdev_ops->vdev_op_leaf &&
2596             (zio->io_type == ZIO_TYPE_READ || zio->io_type == ZIO_TYPE_WRITE)) {

2598                 if (zio->io_type == ZIO_TYPE_READ && vdev_cache_read(zio))
2599                         return (ZIO_PIPELINE_CONTINUE);

2601                 if ((zio = vdev_queue_io(zio)) == NULL)
2602                         return (ZIO_PIPELINE_STOP);

2604                 if (!vdev_accessible(vd, zio)) {
2605                         zio->io_error = SET_ERROR(ENXIO);
2606                         zio_interrupt(zio);
2607                         return (ZIO_PIPELINE_STOP);
2608                 }
2609         }

2611         return (vd->vdev_ops->vdev_op_io_start(zio));
2612 }

2614 static int
2615 zio_vdev_io_done(zio_t *zio)
2616 {
2617         vdev_t *vd = zio->io_vd;
2618         vdev_ops_t *ops = vd ? vd->vdev_ops : &vdev_mirror_ops;
2619         boolean_t unexpected_error = B_FALSE;

2621         if (zio_wait_for_children(zio, ZIO_CHILD_VDEV, ZIO_WAIT_DONE))
2622                 return (ZIO_PIPELINE_STOP);

2624         ASSERT(zio->io_type == ZIO_TYPE_READ || zio->io_type == ZIO_TYPE_WRITE);

2626         if (vd != NULL && vd->vdev_ops->vdev_op_leaf) {

2628                 vdev_queue_io_done(zio);

2630                 if (zio->io_type == ZIO_TYPE_WRITE)
2631                         vdev_cache_write(zio);

2633                 if (zio_injection_enabled && zio->io_error == 0)
2634                         zio->io_error = zio_handle_device_injection(vd,
2635                             zio, EIO);

2637                 if (zio_injection_enabled && zio->io_error == 0)
2638                         zio->io_error = zio_handle_label_injection(zio, EIO);

2640                 if (zio->io_error) {
2641                         if (!vdev_accessible(vd, zio)) {
2642                                 zio->io_error = SET_ERROR(ENXIO);
2643                         } else {
2644                                 unexpected_error = B_TRUE;
2645                         }
2646                 }
2647         }

2649         ops->vdev_op_io_done(zio);

2651         if (unexpected_error)
2652                 VERIFY(vdev_probe(vd, zio) == NULL);

2654         return (ZIO_PIPELINE_CONTINUE);
2655 }
```

```
2657 /*
2658  * For non-raidz ZIOs, we can just copy aside the bad data read from the
2659  * disk, and use that to finish the checksum ereport later.
2660  */
2661 static void
2662 zio_vsd_default_cksum_finish(zio_cksum_report_t *zcr,
2663     const void *good_buf)
2664 {
2665         /* no processing needed */
2666         zfs_ereport_finish_checksum(zcr, good_buf, zcr->zcr_cbdata, B_FALSE);
2667 }

2669 /*ARGSUSED*/
2670 void
2671 zio_vsd_default_cksum_report(zio_t *zio, zio_cksum_report_t *zcr, void *ignored)
2672 {
2673         void *buf = zio_buf_alloc(zio->io_size);

2675         bcopy(zio->io_data, buf, zio->io_size);

2677         zcr->zcr_cbinfo = zio->io_size;
2678         zcr->zcr_cbdata = buf;
2679         zcr->zcr_finish = zio_vsd_default_cksum_finish;
2680         zcr->zcr_free = zio_buf_free;
2681 }

2683 static int
2684 zio_vdev_io_assess(zio_t *zio)
2685 {
2686         vdev_t *vd = zio->io_vd;

2688         if (zio_wait_for_children(zio, ZIO_CHILD_VDEV, ZIO_WAIT_DONE))
2689                 return (ZIO_PIPELINE_STOP);

2691         if (vd == NULL && !(zio->io_flags & ZIO_FLAG_CONFIG_WRITER))
2692                 spa_config_exit(zio->io_spa, SCL_ZIO, zio);

2694         if (zio->io_vsd != NULL) {
2695                 zio->io_vsd_ops->vsd_free(zio);
2696                 zio->io_vsd = NULL;
2697         }

2699         if (zio_injection_enabled && zio->io_error == 0)
2700                 zio->io_error = zio_handle_fault_injection(zio, EIO);

2702         /*
2703          * If the I/O failed, determine whether we should attempt to retry it.
2704          *
2705          * On retry, we cut in line in the issue queue, since we don't want
2706          * compression/checksumming/etc. work to prevent our (cheap) IO reissue.
2707          */
2708         if (zio->io_error && vd == NULL &&
2709             !(zio->io_flags & (ZIO_FLAG_DONT_RETRY | ZIO_FLAG_IO_RETRY))) {
2710                 ASSERT(!(zio->io_flags & ZIO_FLAG_DONT_QUEUE)); /* not a leaf */
2711                 ASSERT(!(zio->io_flags & ZIO_FLAG_IO_BYPASS));  /* not a leaf */
2712                 zio->io_error = 0;
2713                 zio->io_flags |= ZIO_FLAG_IO_RETRY |
2714                     ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE;
2715                 zio->io_stage = ZIO_STAGE_VDEV_IO_START >> 1;
2716                 zio_taskq_dispatch(zio, ZIO_TASKQ_ISSUE,
2717                     zio_requeue_io_start_cut_in_line);
2718                 return (ZIO_PIPELINE_STOP);
2719         }

2721         /*
```



```
2722          * If we got an error on a leaf device, convert it to ENXIO
2723          * if the device is not accessible at all.
2724          */
2725         if (zio->io_error && vd != NULL && vd->vdev_ops->vdev_op_leaf &&
2726             !vdev_accessible(vd, zio))
2727                 zio->io_error = SET_ERROR(ENXIO);

2729         /*
2730          * If we can't write to an interior vdev (mirror or RAID-Z),
2731          * set vdev_cant_write so that we stop trying to allocate from it.
2732          */
2733         if (zio->io_error == ENXIO && zio->io_type == ZIO_TYPE_WRITE &&
2734             vd != NULL && !vd->vdev_ops->vdev_op_leaf) {
2735                 vd->vdev_cant_write = B_TRUE;
2736         }

2738         if (zio->io_error)
2739                 zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;

2741         if (vd != NULL && vd->vdev_ops->vdev_op_leaf &&
2742             zio->io_physdone != NULL) {
2743                 ASSERT(!(zio->io_flags & ZIO_FLAG_DELEGATED));
2744                 ASSERT(zio->io_child_type == ZIO_CHILD_VDEV);
2745                 zio->io_physdone(zio->io_logical);
2746         }

2748         return (ZIO_PIPELINE_CONTINUE);
2749 }

2751 void
2752 zio_vdev_io_reissue(zio_t *zio)
2753 {
2754         ASSERT(zio->io_stage == ZIO_STAGE_VDEV_IO_START);
2755         ASSERT(zio->io_error == 0);

2757         zio->io_stage >>= 1;
2758 }

2760 void
2761 zio_vdev_io_redone(zio_t *zio)
2762 {
2763         ASSERT(zio->io_stage == ZIO_STAGE_VDEV_IO_DONE);

2765         zio->io_stage >>= 1;
2766 }

2768 void
2769 zio_vdev_io_bypass(zio_t *zio)
2770 {
2771         ASSERT(zio->io_stage == ZIO_STAGE_VDEV_IO_START);
2772         ASSERT(zio->io_error == 0);

2774         zio->io_flags |= ZIO_FLAG_IO_BYPASS;
2775         zio->io_stage = ZIO_STAGE_VDEV_IO_ASSESS >> 1;
2776 }

2778 /*
2779  * ==========================================================================
2780  * Generate and verify checksums
2781  * ==========================================================================
2782  */
2783 static int
2784 zio_checksum_generate(zio_t *zio)
2785 {
2786         blkptr_t *bp = zio->io_bp;
2787         enum zio_checksum checksum;
```

```
2789         if (bp == NULL) {
2790                 /*
2791                  * This is zio_write_phys().
2792                  * We're either generating a label checksum, or none at all.
2793                  */
2794                 checksum = zio->io_prop.zp_checksum;

2796                 if (checksum == ZIO_CHECKSUM_OFF)
2797                         return (ZIO_PIPELINE_CONTINUE);

2799                 ASSERT(checksum == ZIO_CHECKSUM_LABEL);
2800         } else {
2801                 if (BP_IS_GANG(bp) && zio->io_child_type == ZIO_CHILD_GANG) {
2802                         ASSERT(!IO_IS_ALLOCATING(zio));
2803                         checksum = ZIO_CHECKSUM_GANG_HEADER;
2804                 } else {
2805                         checksum = BP_GET_CHECKSUM(bp);
2806                 }
2807         }

2809         zio_checksum_compute(zio, checksum, zio->io_data, zio->io_size);

2811         return (ZIO_PIPELINE_CONTINUE);
2812 }

2814 static int
2815 zio_checksum_verify(zio_t *zio)
2816 {
2817         zio_bad_cksum_t info;
2818         blkptr_t *bp = zio->io_bp;
2819         int error;

2821         ASSERT(zio->io_vd != NULL);

2823         if (bp == NULL) {
2824                 /*
2825                  * This is zio_read_phys().
2826                  * We're either verifying a label checksum, or nothing at all.
2827                  */
2828                 if (zio->io_prop.zp_checksum == ZIO_CHECKSUM_OFF)
2829                         return (ZIO_PIPELINE_CONTINUE);

2831                 ASSERT(zio->io_prop.zp_checksum == ZIO_CHECKSUM_LABEL);
2832         }

2834         if ((error = zio_checksum_error(zio, &info)) != 0) {
2835                 zio->io_error = error;
2836                 if (!(zio->io_flags & ZIO_FLAG_SPECULATIVE)) {
2837                         zfs_ereport_start_checksum(zio->io_spa,
2838                             zio->io_vd, zio, zio->io_offset,
2839                             zio->io_size, NULL, &info);
2840                 }
2841         }

2843         return (ZIO_PIPELINE_CONTINUE);
2844 }

2846 /*
2847  * Called by RAID-Z to ensure we don't compute the checksum twice.
2848  */
2849 void
2850 zio_checksum_verified(zio_t *zio)
2851 {
2852         zio->io_pipeline &= ~ZIO_STAGE_CHECKSUM_VERIFY;
2853 }
```



```
2855 /*
2856  * ==========================================================================
2857  * Error rank.  Error are ranked in the order 0, ENXIO, ECKSUM, EIO, other.
2858  * An error of 0 indicates success.  ENXIO indicates whole-device failure,
2859  * which may be transient (e.g. unplugged) or permament.  ECKSUM and EIO
2860  * indicate errors that are specific to one I/O, and most likely permanent.
2861  * Any other error is presumed to be worse because we weren't expecting it.
2862  * ==========================================================================
2863  */
2864 int
2865 zio_worst_error(int e1, int e2)
2866 {
2867         static int zio_error_rank[] = { 0, ENXIO, ECKSUM, EIO };
2868         int r1, r2;

2870         for (r1 = 0; r1 < sizeof (zio_error_rank) / sizeof (int); r1++)
2871                 if (e1 == zio_error_rank[r1])
2872                         break;

2874         for (r2 = 0; r2 < sizeof (zio_error_rank) / sizeof (int); r2++)
2875                 if (e2 == zio_error_rank[r2])
2876                         break;

2878         return (r1 > r2 ? e1 : e2);
2879 }

2881 /*
2882  * ==========================================================================
2883  * I/O completion
2884  * ==========================================================================
2885  */
2886 static int
2887 zio_ready(zio_t *zio)
2888 {
2889         blkptr_t *bp = zio->io_bp;
2890         zio_t *pio, *pio_next;

2892         if (zio_wait_for_children(zio, ZIO_CHILD_GANG, ZIO_WAIT_READY) ||
2893             zio_wait_for_children(zio, ZIO_CHILD_DDT, ZIO_WAIT_READY))
2894                 return (ZIO_PIPELINE_STOP);

2896         if (zio->io_ready) {
2897                 ASSERT(IO_IS_ALLOCATING(zio));
2898                 ASSERT(bp->blk_birth == zio->io_txg || BP_IS_HOLE(bp) ||
2899                     (zio->io_flags & ZIO_FLAG_NOPWRITE));
2900                 ASSERT(zio->io_children[ZIO_CHILD_GANG][ZIO_WAIT_READY] == 0);

2902                 zio->io_ready(zio);
2903         }

2905         if (bp != NULL && bp != &zio->io_bp_copy)
2906                 zio->io_bp_copy = *bp;

2908         if (zio->io_error)
2909                 zio->io_pipeline = ZIO_INTERLOCK_PIPELINE;

2911         mutex_enter(&zio->io_lock);
2912         zio->io_state[ZIO_WAIT_READY] = 1;
2913         pio = zio_walk_parents(zio);
2914         mutex_exit(&zio->io_lock);

2916         /*
2917          * As we notify zio's parents, new parents could be added.
2918          * New parents go to the head of zio's io_parent_list, however,
2919          * so we will (correctly) not notify them.  The remainder of zio's
```

```
2920          * io_parent_list, from 'pio_next' onward, cannot change because
2921          * all parents must wait for us to be done before they can be done.
2922          */
2923         for (; pio != NULL; pio = pio_next) {
2924                 pio_next = zio_walk_parents(zio);
2925                 zio_notify_parent(pio, zio, ZIO_WAIT_READY);
2926         }

2928         if (zio->io_flags & ZIO_FLAG_NODATA) {
2929                 if (BP_IS_GANG(bp)) {
2930                         zio->io_flags &= ~ZIO_FLAG_NODATA;
2931                 } else {
2932                         ASSERT((uintptr_t)zio->io_data < SPA_MAXBLOCKSIZE);
2933                         zio->io_pipeline &= ~ZIO_VDEV_IO_STAGES;
2934                 }
2935         }

2937         if (zio_injection_enabled &&
2938             zio->io_spa->spa_syncing_txg == zio->io_txg)
2939                 zio_handle_ignored_writes(zio);

2941         return (ZIO_PIPELINE_CONTINUE);
2942 }

2944 static int
2945 zio_done(zio_t *zio)
2946 {
2947         spa_t *spa = zio->io_spa;
2948         zio_t *lio = zio->io_logical;
2949         blkptr_t *bp = zio->io_bp;
2950         vdev_t *vd = zio->io_vd;
2951         uint64_t psize = zio->io_size;
2952         zio_t *pio, *pio_next;

2954         /*
2955          * If our children haven't all completed,
2956          * wait for them and then repeat this pipeline stage.
2957          */
2958         if (zio_wait_for_children(zio, ZIO_CHILD_VDEV, ZIO_WAIT_DONE) ||
2959             zio_wait_for_children(zio, ZIO_CHILD_GANG, ZIO_WAIT_DONE) ||
2960             zio_wait_for_children(zio, ZIO_CHILD_DDT, ZIO_WAIT_DONE) ||
2961             zio_wait_for_children(zio, ZIO_CHILD_LOGICAL, ZIO_WAIT_DONE))
2962                 return (ZIO_PIPELINE_STOP);

2964         for (int c = 0; c < ZIO_CHILD_TYPES; c++)
2965                 for (int w = 0; w < ZIO_WAIT_TYPES; w++)
2966                         ASSERT(zio->io_children[c][w] == 0);

2968         if (bp != NULL && !BP_IS_EMBEDDED(bp)) {
2969                 ASSERT(bp->blk_pad[0] == 0);
2970                 ASSERT(bp->blk_pad[1] == 0);
2971                 ASSERT(bcmp(bp, &zio->io_bp_copy, sizeof (blkptr_t)) == 0 ||
2972                     (bp == zio_unique_parent(zio)->io_bp));
2973                 if (zio->io_type == ZIO_TYPE_WRITE && !BP_IS_HOLE(bp) &&
2974                     zio->io_bp_override == NULL &&
2975                     !(zio->io_flags & ZIO_FLAG_IO_REPAIR)) {
2976                         ASSERT(!BP_SHOULD_BYTESWAP(bp));
2977                         ASSERT3U(zio->io_prop.zp_copies, <=, BP_GET_NDVAS(bp));
2978                         ASSERT(BP_COUNT_GANG(bp) == 0 ||
2979                             (BP_COUNT_GANG(bp) == BP_GET_NDVAS(bp)));
2980                 }
2981                 if (zio->io_flags & ZIO_FLAG_NOPWRITE)
2982                         VERIFY(BP_EQUAL(bp, &zio->io_bp_orig));
2983         }

2985         /*
```



```
2986          * If there were child vdev/gang/ddt errors, they apply to us now.
2987          */
2988         zio_inherit_child_errors(zio, ZIO_CHILD_VDEV);
2989         zio_inherit_child_errors(zio, ZIO_CHILD_GANG);
2990         zio_inherit_child_errors(zio, ZIO_CHILD_DDT);

2992         /*
2993          * If the I/O on the transformed data was successful, generate any
2994          * checksum reports now while we still have the transformed data.
2995          */
2996         if (zio->io_error == 0) {
2997                 while (zio->io_cksum_report != NULL) {
2998                         zio_cksum_report_t *zcr = zio->io_cksum_report;
2999                         uint64_t align = zcr->zcr_align;
3000                         uint64_t asize = P2ROUNDUP(psize, align);
3001                         char *abuf = zio->io_data;

3003                         if (asize != psize) {
3004                                 abuf = zio_buf_alloc(asize);
3005                                 bcopy(zio->io_data, abuf, psize);
3006                                 bzero(abuf + psize, asize - psize);
3007                         }

3009                         zio->io_cksum_report = zcr->zcr_next;
3010                         zcr->zcr_next = NULL;
3011                         zcr->zcr_finish(zcr, abuf);
3012                         zfs_ereport_free_checksum(zcr);

3014                         if (asize != psize)
3015                                 zio_buf_free(abuf, asize);
3016                 }
3017         }

3019         zio_pop_transforms(zio);        /* note: may set zio->io_error */

3021         vdev_stat_update(zio, psize);

3023         if (zio->io_error) {
3024                 /*
3025                  * If this I/O is attached to a particular vdev,
3026                  * generate an error message describing the I/O failure
3027                  * at the block level.  We ignore these errors if the
3028                  * device is currently unavailable.
3029                  */
3030                 if (zio->io_error != ECKSUM && vd != NULL && !vdev_is_dead(vd))
3031                         zfs_ereport_post(FM_EREPORT_ZFS_IO, spa, vd, zio, 0, 0);

3033                 if ((zio->io_error == EIO || !(zio->io_flags &
3034                     (ZIO_FLAG_SPECULATIVE | ZIO_FLAG_DONT_PROPAGATE))) &&
3035                     zio == lio) {
3036                         /*
3037                          * For logical I/O requests, tell the SPA to log the
3038                          * error and generate a logical data ereport.
3039                          */
3040                         spa_log_error(spa, zio);
3041                         zfs_ereport_post(FM_EREPORT_ZFS_DATA, spa, NULL, zio,
3042                             0, 0);
3043                 }
3044         }

3046         if (zio->io_error && zio == lio) {
3047                 /*
3048                  * Determine whether zio should be reexecuted.  This will
3049                  * propagate all the way to the root via zio_notify_parent().
3050                  */
3051                 ASSERT(vd == NULL && bp != NULL);
```

```
3052                ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);

3054                if (IO_IS_ALLOCATING(zio) &&
3055                    !(zio->io_flags & ZIO_FLAG_CANFAIL)) {
3056                        if (zio->io_error != ENOSPC)
3057                                zio->io_reexecute |= ZIO_REEXECUTE_NOW;
3058                        else
3059                                zio->io_reexecute |= ZIO_REEXECUTE_SUSPEND;
3060                }

3062                if ((zio->io_type == ZIO_TYPE_READ ||
3063                    zio->io_type == ZIO_TYPE_FREE) &&
3064                    !(zio->io_flags & ZIO_FLAG_SCAN_THREAD) &&
3065                    zio->io_error == ENXIO &&
3066                    spa_load_state(spa) == SPA_LOAD_NONE &&
3067                    spa_get_failmode(spa) != ZIO_FAILURE_MODE_CONTINUE)
3068                        zio->io_reexecute |= ZIO_REEXECUTE_SUSPEND;

3070                if (!(zio->io_flags & ZIO_FLAG_CANFAIL) && !zio->io_reexecute)
3071                        zio->io_reexecute |= ZIO_REEXECUTE_SUSPEND;

3073                /*
3074                 * Here is a possibly good place to attempt to do
3075                 * either combinatorial reconstruction or error correction
3076                 * based on checksums.  It also might be a good place
3077                 * to send out preliminary ereports before we suspend
3078                 * processing.
3079                 */
3080        }

3082        /*
3083         * If there were logical child errors, they apply to us now.
3084         * We defer this until now to avoid conflating logical child
3085         * errors with errors that happened to the zio itself when
3086         * updating vdev stats and reporting FMA events above.
3087         */
3088        zio_inherit_child_errors(zio, ZIO_CHILD_LOGICAL);

3090        if ((zio->io_error || zio->io_reexecute) &&
3091            IO_IS_ALLOCATING(zio) && zio->io_gang_leader == zio &&
3092            !(zio->io_flags & (ZIO_FLAG_IO_REWRITE | ZIO_FLAG_NOPWRITE)))
3093                zio_dva_unallocate(zio, zio->io_gang_tree, bp);

3095        zio_gang_tree_free(&zio->io_gang_tree);

3097        /*
3098         * Godfather I/Os should never suspend.
3099         */
3100        if ((zio->io_flags & ZIO_FLAG_GODFATHER) &&
3101            (zio->io_reexecute & ZIO_REEXECUTE_SUSPEND))
3102                zio->io_reexecute = 0;

3104        if (zio->io_reexecute) {
3105                /*
3106                 * This is a logical I/O that wants to reexecute.
3107                 *
3108                 * Reexecute is top-down.  When an i/o fails, if it's not
3109                 * the root, it simply notifies its parent and sticks around.
3110                 * The parent, seeing that it still has children in zio_done(),
3111                 * does the same.  This percolates all the way up to the root.
3112                 * The root i/o will reexecute or suspend the entire tree.
3113                 *
3114                 * This approach ensures that zio_reexecute() honors
3115                 * all the original i/o dependency relationships, e.g.
3116                 * parents not executing until children are ready.
3117                 */
```



```
3118                ASSERT(zio->io_child_type == ZIO_CHILD_LOGICAL);

3120                zio->io_gang_leader = NULL;

3122                mutex_enter(&zio->io_lock);
3123                zio->io_state[ZIO_WAIT_DONE] = 1;
3124                mutex_exit(&zio->io_lock);

3126                /*
3127                 * "The Godfather" I/O monitors its children but is
3128                 * not a true parent to them. It will track them through
3129                 * the pipeline but severs its ties whenever they get into
3130                 * trouble (e.g. suspended). This allows "The Godfather"
3131                 * I/O to return status without blocking.
3132                 */
3133                for (pio = zio_walk_parents(zio); pio != NULL; pio = pio_next) {
3134                        zio_link_t *zl = zio->io_walk_link;
3135                        pio_next = zio_walk_parents(zio);

3137                        if ((pio->io_flags & ZIO_FLAG_GODFATHER) &&
3138                            (zio->io_reexecute & ZIO_REEXECUTE_SUSPEND)) {
3139                                zio_remove_child(pio, zio, zl);
3140                                zio_notify_parent(pio, zio, ZIO_WAIT_DONE);
3141                        }
3142                }

3144                if ((pio = zio_unique_parent(zio)) != NULL) {
3145                        /*
3146                         * We're not a root i/o, so there's nothing to do
3147                         * but notify our parent.  Don't propagate errors
3148                         * upward since we haven't permanently failed yet.
3149                         */
3150                        ASSERT(!(zio->io_flags & ZIO_FLAG_GODFATHER));
3151                        zio->io_flags |= ZIO_FLAG_DONT_PROPAGATE;
3152                        zio_notify_parent(pio, zio, ZIO_WAIT_DONE);
3153                } else if (zio->io_reexecute & ZIO_REEXECUTE_SUSPEND) {
3154                        /*
3155                         * We'd fail again if we reexecuted now, so suspend
3156                         * until conditions improve (e.g. device comes online).
3157                         */
3158                        zio_suspend(spa, zio);
3159                } else {
3160                        /*
3161                         * Reexecution is potentially a huge amount of work.
3162                         * Hand it off to the otherwise-unused claim taskq.
3163                         */
3164                        ASSERT(zio->io_tqent.tqent_next == NULL);
3165                        spa_taskq_dispatch_ent(spa, ZIO_TYPE_CLAIM,
3166                            ZIO_TASKQ_ISSUE, (task_func_t *)zio_reexecute, zio,
3167                            0, &zio->io_tqent);
3168                }
3169                return (ZIO_PIPELINE_STOP);
3170        }

3172        ASSERT(zio->io_child_count == 0);
3173        ASSERT(zio->io_reexecute == 0);
3174        ASSERT(zio->io_error == 0 || (zio->io_flags & ZIO_FLAG_CANFAIL));

3176        /*
3177         * Report any checksum errors, since the I/O is complete.
3178         */
3179        while (zio->io_cksum_report != NULL) {
3180                zio_cksum_report_t *zcr = zio->io_cksum_report;
3181                zio->io_cksum_report = zcr->zcr_next;
3182                zcr->zcr_next = NULL;
3183                zcr->zcr_finish(zcr, NULL);
```

```
3184                    zfs_ereport_free_checksum(zcr);
3185            }
3186
3187            /*
3188             * It is the responsibility of the done callback to ensure that this
3189             * particular zio is no longer discoverable for adoption, and as
3190             * such, cannot acquire any new parents.
3191             */
3192            if (zio->io_done)
3193                    zio->io_done(zio);
3194
3195            mutex_enter(&zio->io_lock);
3196            zio->io_state[ZIO_WAIT_DONE] = 1;
3197            mutex_exit(&zio->io_lock);
3198
3199            for (pio = zio_walk_parents(zio); pio != NULL; pio = pio_next) {
3200                    zio_link_t *zl = zio->io_walk_link;
3201                    pio_next = zio_walk_parents(zio);
3202                    zio_remove_child(pio, zio, zl);
3203                    zio_notify_parent(pio, zio, ZIO_WAIT_DONE);
3204            }
3205
3206            if (zio->io_waiter != NULL) {
3207                    mutex_enter(&zio->io_lock);
3208                    zio->io_executor = NULL;
3209                    cv_broadcast(&zio->io_cv);
3210                    mutex_exit(&zio->io_lock);
3211            } else {
3212                    zio_destroy(zio);
3213            }
3214
3215            return (ZIO_PIPELINE_STOP);
3216    }
3217
3218    /*
3219     * ==========================================================================
3220     * I/O pipeline definition
3221     * ==========================================================================
3222     */
3223    static zio_pipe_stage_t *zio_pipeline[] = {
3224            NULL,
3225            zio_read_bp_init,
3226            zio_free_bp_init,
3227            zio_issue_async,
3228            zio_write_bp_init,
3229            zio_checksum_generate,
3230            zio_nop_write,
3231            zio_ddt_read_start,
3232            zio_ddt_read_done,
3233            zio_ddt_write,
3234            zio_ddt_free,
3235            zio_gang_assemble,
3236            zio_gang_issue,
3237            zio_dva_allocate,
3238            zio_dva_free,
3239            zio_dva_claim,
3240            zio_ready,
3241            zio_vdev_io_start,
3242            zio_vdev_io_done,
3243            zio_vdev_io_assess,
3244            zio_checksum_verify,
3245            zio_done
3246    };
3247
3248    /* dnp is the dnode for zb1->zb_object */
3249    boolean_t
```



```
3250    zbookmark_is_before(const dnode_phys_t *dnp, const zbookmark_phys_t *zb1,
3251        const zbookmark_phys_t *zb2)
3252    {
3253            uint64_t zb1nextL0, zb2thisobj;
3254
3255            ASSERT(zb1->zb_objset == zb2->zb_objset);
3256            ASSERT(zb2->zb_level == 0);
3257
3258            /* The objset_phys_t isn't before anything. */
3259            if (dnp == NULL)
3260                    return (B_FALSE);
3261
3262            zb1nextL0 = (zb1->zb_blkid + 1) <<
3263                ((zb1->zb_level) * (dnp->dn_indblkshift - SPA_BLKPTRSHIFT));
3264
3265            zb2thisobj = zb2->zb_object ? zb2->zb_object :
3266                zb2->zb_blkid << (DNODE_BLOCK_SHIFT - DNODE_SHIFT);
3267
3268            if (zb1->zb_object == DMU_META_DNODE_OBJECT) {
3269                    uint64_t nextobj = zb1nextL0 *
3270                        (dnp->dn_datablkszsec << SPA_MINBLOCKSHIFT) >> DNODE_SHIFT;
3271                    return (nextobj <= zb2thisobj);
3272            }
3273
3274            if (zb1->zb_object < zb2thisobj)
3275                    return (B_TRUE);
3276            if (zb1->zb_object > zb2thisobj)
3277                    return (B_FALSE);
3278            if (zb2->zb_object == DMU_META_DNODE_OBJECT)
3279                    return (B_FALSE);
3280            return (zb1nextL0 <= zb2->zb_blkid);
3281    }
```

```
**********************************************************
    4410 Tue Oct 28 11:57:20 2014
new/usr/src/uts/common/sys/filio.h
Possibility to physically reserve space without writing leaf blocks
**********************************************************
```

```
   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 2007 Sun Microsystems, Inc.  All rights reserved.
  23  * Use is subject to license terms.
  24  */

  26 /*      Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T     */
  27 /*        All Rights Reserved   */

  29 /*
  30  * University Copyright- Copyright (c) 1982, 1986, 1988
  31  * The Regents of the University of California
  32  * All Rights Reserved
  33  *
  34  * University Acknowledgment- Portions of this document are derived from
  35  * software developed by the University of California, Berkeley, and its
  36  * contributors.
  37  */

  39 #ifndef _SYS_FILIO_H
  40 #define _SYS_FILIO_H

  42 #pragma ident   "%Z%%M% %I%     %E% SMI"

  44 /*
  45  * General file ioctl definitions.
  46  */

  48 #include <sys/ioccom.h>

  50 #ifdef __cplusplus
  51 extern "C" {
  52 #endif

  54 #define FIOCLEX         _IO('f', 1)             /* set exclusive use on fd */
  55 #define FIONCLEX        _IO('f', 2)             /* remove exclusive use */
  56 /* another local */
  57 #define FIONREAD        _IOR('f', 127, int)     /* get # bytes to read */
  58 #define FIONBIO         _IOW('f', 126, int)     /* set/clear non-blocking i/o */
  59 #define FIOASYNC        _IOW('f', 125, int)     /* set/clear async i/o */
  60 #define FIOSETOWN       _IOW('f', 124, int)     /* set owner */
  61 #define FIOGETOWN       _IOR('f', 123, int)     /* get owner */
```



```
  63 /*
  64  * ioctl's for Online: DiskSuite.
  65  * WARNING - the support for these ioctls may be withdrawn
  66  * in future OS releases.
  67  */
  68 #define _FIOLFS         _IO('f', 64)            /* file system lock */
  69 #define _FIOLFSS        _IO('f', 65)            /* file system lock status */
  70 #define _FIOFFS         _IO('f', 66)            /* file system flush */
  71 #define _FIOAI          _FIOOBSOLETE67          /* get allocation info is */
  72 #define _FIOOBSOLETE67  _IO('f', 67)            /* obsolete and unsupported */
  73 #define _FIOSATIME      _IO('f', 68)            /* set atime */
  74 #define _FIOSDIO        _IO('f', 69)            /* set delayed io */
  75 #define _FIOGDIO        _IO('f', 70)            /* get delayed io */
  76 #define _FIOIO          _IO('f', 71)            /* inode open */
  77 #define _FIOISLOG       _IO('f', 72)            /* disksuite/ufs protocol */
  78 #define _FIOISLOGOK     _IO('f', 73)            /* disksuite/ufs protocol */
  79 #define _FIOLOGRESET    _IO('f', 74)            /* disksuite/ufs protocol */

  81 /*
  82  * Contract-private ioctl()
  83  */
  84 #define _FIOISBUSY      _IO('f', 75)            /* networker/ufs protocol */
  85 #define _FIODIRECTIO    _IO('f', 76)            /* directio */
  86 #define _FIOTUNE        _IO('f', 77)            /* tuning */

  88 /*
  89  * WARNING: These 'f' ioctls are also defined in sys/fs/cachefs_fs.h
  90  * It currently defines 78-86.
  91  */

  93 /*
  94  * Internal Logging UFS
  95  */
  96 #define _FIOLOGENABLE   _IO('f', 87)            /* logging/ufs protocol */
  97 #define _FIOLOGDISABLE  _IO('f', 88)            /* logging/ufs protocol */

  99 /*
 100  * File system snapshot ioctls (see sys/fs/ufs_snap.h)
 101  * (there is another snapshot ioctl, _FIOSNAPSHOTCREATE_MULTI,
 102  * defined farther down in this file.)
 103  */
 104 #define _FIOSNAPSHOTCREATE      _IO('f', 89)    /* create a snapshot */
 105 #define _FIOSNAPSHOTDELETE      _IO('f', 90)    /* delete a snapshot */

 107 /*
 108  * Return the current superblock of size SBSIZE
 109  */
 110 #define _FIOGETSUPERBLOCK       _IO('f', 91)

 112 /*
 113  * Contract private ioctl
 114  */
 115 #define _FIOGETMAXPHYS          _IO('f', 92)

 117 /*
 118  * TSufs support
 119  */
 120 #define _FIO_SET_LUFS_DEBUG     _IO('f', 93) /* set lufs_debug */
 121 #define _FIO_SET_LUFS_ERROR     _IO('f', 94) /* set a lufs error */
 122 #define _FIO_GET_TOP_STATS      _IO('f', 95) /* get lufs tranaction stats */

 124 /*
 125  * create a snapshot with multiple backing files
 126  */
 127 #define _FIOSNAPSHOTCREATE_MULTI        _IO('f', 96)
```

```
129 /*
130  * handle lseek SEEK_DATA and SEEK_HOLE for holey file knowledge
131  */
132 #define _FIO_SEEK_DATA          _IO('f', 97) /* SEEK_DATA */
133 #define _FIO_SEEK_HOLE          _IO('f', 98) /* SEEK_HOLE */

135 /*
136  * boot archive compression
137  */
138 #define _FIO_COMPRESSED         _IO('f', 99) /* mark file as compressed */
139 #define _FIO_RESERVE_SPACE      _IO('f', 100) /* Reserve space */
140 #endif /* ! codereview */

142 #ifdef  __cplusplus
143 }
144 #endif

146 #endif  /* _SYS_FILIO_H */
```