```
**********************************************************
   50306 Wed May  6 08:47:27 2015
new/usr/src/uts/common/fs/zfs/dmu_objset.c
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
/*
 * 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 (c) 2013 by Saso Kiselkov. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 * Copyright (c) 2014 Spectra Logic Corporation, All rights reserved.
 * Copyright 2015 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2015, STRATO AG, Inc. All rights reserved.
#endif /* ! codereview */
 */

/* Portions Copyright 2010 Robert Milkowski */

#include <sys/cred.h>
#include <sys/zfs_context.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_deleg.h>
#include <sys/dnode.h>
#include <sys/dbuf.h>
#include <sys/zvol.h>
#include <sys/dmu_tx.h>
#include <sys/zap.h>
#include <sys/zil.h>
#include <sys/dmu_impl.h>
#include <sys/zfs_ioctl.h>
#include <sys/sa.h>
```

```
#include <sys/zfs_onexit.h>
#include <sys/dsl_destroy.h>
#include <sys/vdev.h>
#endif /* ! codereview */

/*
 * Needed to close a window in dnode_move() that allows the objset to be freed
 * before it can be safely accessed.
 */
krwlock_t os_lock;

/*
 * Tunable to overwrite the maximum number of threads for the parallization
 * of dmu_objset_find_dp, needed to speed up the import of pools with many
 * datasets.
 * Default is 4 times the number of leaf vdevs.
 */
int dmu_find_threads = 0;

static void dmu_objset_find_dp_cb(void *arg);

#endif /* ! codereview */
void
dmu_objset_init(void)
{
        rw_init(&os_lock, NULL, RW_DEFAULT, NULL);
}

void
dmu_objset_fini(void)
{
        rw_destroy(&os_lock);
}

spa_t *
dmu_objset_spa(objset_t *os)
{
        return (os->os_spa);
}

zilog_t *
dmu_objset_zil(objset_t *os)
{
        return (os->os_zil);
}

dsl_pool_t *
dmu_objset_pool(objset_t *os)
{
        dsl_dataset_t *ds;

        if ((ds = os->os_dsl_dataset) != NULL && ds->ds_dir)
                return (ds->ds_dir->dd_pool);
        else
                return (spa_get_dsl(os->os_spa));
}

dsl_dataset_t *
dmu_objset_ds(objset_t *os)
{
        return (os->os_dsl_dataset);
}

dmu_objset_type_t
dmu_objset_type(objset_t *os)
{
        return (os->os_phys->os_type);
}

void
dmu_objset_name(objset_t *os, char *buf)
{
        dsl_dataset_name(os->os_dsl_dataset, buf);
}

uint64_t
dmu_objset_id(objset_t *os)
{
        dsl_dataset_t *ds = os->os_dsl_dataset;

        return (ds ? ds->ds_object : 0);
}

zfs_sync_type_t
dmu_objset_syncprop(objset_t *os)
{
        return (os->os_sync);
}

zfs_logbias_op_t
dmu_objset_logbias(objset_t *os)
{
        return (os->os_logbias);
}

static void
checksum_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        /*
         * Inheritance should have been done by now.
         */
        ASSERT(newval != ZIO_CHECKSUM_INHERIT);

        os->os_checksum = zio_checksum_select(newval, ZIO_CHECKSUM_ON_VALUE);
}

static void
compression_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        /*
         * Inheritance and range checking should have been done by now.
         */
        ASSERT(newval != ZIO_COMPRESS_INHERIT);

        os->os_compress = zio_compress_select(os->os_spa, newval,
            ZIO_COMPRESS_ON);
}

static void
copies_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        /*
         * Inheritance and range checking should have been done by now.
         */
        ASSERT(newval > 0);
        ASSERT(newval <= spa_max_replication(os->os_spa));
```

```

        os->os_copies = newval;
}

static void
dedup_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;
        spa_t *spa = os->os_spa;
        enum zio_checksum checksum;

        /*
         * Inheritance should have been done by now.
         */
        ASSERT(newval != ZIO_CHECKSUM_INHERIT);

        checksum = zio_checksum_dedup_select(spa, newval, ZIO_CHECKSUM_OFF);

        os->os_dedup_checksum = checksum & ZIO_CHECKSUM_MASK;
        os->os_dedup_verify = !!(checksum & ZIO_CHECKSUM_VERIFY);
}

static void
primary_cache_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        /*
         * Inheritance and range checking should have been done by now.
         */
        ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
            newval == ZFS_CACHE_METADATA);

        os->os_primary_cache = newval;
}

static void
secondary_cache_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        /*
         * Inheritance and range checking should have been done by now.
         */
        ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
            newval == ZFS_CACHE_METADATA);

        os->os_secondary_cache = newval;
}

static void
sync_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        /*
         * Inheritance and range checking should have been done by now.
         */
        ASSERT(newval == ZFS_SYNC_STANDARD || newval == ZFS_SYNC_ALWAYS ||
            newval == ZFS_SYNC_DISABLED);

        os->os_sync = newval;
        if (os->os_zil)
                zil_set_sync(os->os_zil, newval);
}
```

```
static void
redundant_metadata_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        /*
         * Inheritance and range checking should have been done by now.
         */
        ASSERT(newval == ZFS_REDUNDANT_METADATA_ALL ||
            newval == ZFS_REDUNDANT_METADATA_MOST);

        os->os_redundant_metadata = newval;
}

static void
logbias_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        ASSERT(newval == ZFS_LOGBIAS_LATENCY ||
            newval == ZFS_LOGBIAS_THROUGHPUT);
        os->os_logbias = newval;
        if (os->os_zil)
                zil_set_logbias(os->os_zil, newval);
}

static void
recordsize_changed_cb(void *arg, uint64_t newval)
{
        objset_t *os = arg;

        os->os_recordsize = newval;
}

void
dmu_objset_byteswap(void *buf, size_t size)
{
        objset_phys_t *osp = buf;

        ASSERT(size == OBJSET_OLD_PHYS_SIZE || size == sizeof (objset_phys_t));
        dnode_byteswap(&osp->os_meta_dnode);
        byteswap_uint64_array(&osp->os_zil_header, sizeof (zil_header_t));
        osp->os_type = BSWAP_64(osp->os_type);
        osp->os_flags = BSWAP_64(osp->os_flags);
        if (size == sizeof (objset_phys_t)) {
                dnode_byteswap(&osp->os_userused_dnode);
                dnode_byteswap(&osp->os_groupused_dnode);
        }
}

int
dmu_objset_open_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
    objset_t **osp)
{
        objset_t *os;
        int i, err;

        ASSERT(ds == NULL || MUTEX_HELD(&ds->ds_opening_lock));

        os = kmem_zalloc(sizeof (objset_t), KM_SLEEP);
        os->os_dsl_dataset = ds;
        os->os_spa = spa;
        os->os_rootbp = bp;
        if (!BP_IS_HOLE(os->os_rootbp)) {
                arc_flags_t aflags = ARC_FLAG_WAIT;
                zbookmark_phys_t zb;
```

```
                SET_BOOKMARK(&zb, ds ? ds->ds_object : DMU_META_OBJSET,
                    ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);

                if (DMU_OS_IS_L2CACHEABLE(os))
                        aflags |= ARC_FLAG_L2CACHE;
                if (DMU_OS_IS_L2COMPRESSIBLE(os))
                        aflags |= ARC_FLAG_L2COMPRESS;

                dprintf_bp(os->os_rootbp, "reading %s", "");
                err = arc_read(NULL, spa, os->os_rootbp,
                    arc_getbuf_func, &os->os_phys_buf,
                    ZIO_PRIORITY_SYNC_READ, ZIO_FLAG_CANFAIL, &aflags, &zb);
                if (err != 0) {
                        kmem_free(os, sizeof (objset_t));
                        /* convert checksum errors into IO errors */
                        if (err == ECKSUM)
                                err = SET_ERROR(EIO);
                        return (err);
                }

                /* Increase the blocksize if we are permitted. */
                if (spa_version(spa) >= SPA_VERSION_USERSPACE &&
                    arc_buf_size(os->os_phys_buf) < sizeof (objset_phys_t)) {
                        arc_buf_t *buf = arc_buf_alloc(spa,
                            sizeof (objset_phys_t), &os->os_phys_buf,
                            ARC_BUFC_METADATA);
                        bzero(buf->b_data, sizeof (objset_phys_t));
                        bcopy(os->os_phys_buf->b_data, buf->b_data,
                            arc_buf_size(os->os_phys_buf));
                        (void) arc_buf_remove_ref(os->os_phys_buf,
                            &os->os_phys_buf);
                        os->os_phys_buf = buf;
                }

                os->os_phys = os->os_phys_buf->b_data;
                os->os_flags = os->os_phys->os_flags;
        } else {
                int size = spa_version(spa) >= SPA_VERSION_USERSPACE ?
                    sizeof (objset_phys_t) : OBJSET_OLD_PHYS_SIZE;
                os->os_phys_buf = arc_buf_alloc(spa, size,
                    &os->os_phys_buf, ARC_BUFC_METADATA);
                os->os_phys = os->os_phys_buf->b_data;
                bzero(os->os_phys, size);
        }

        /*
         * Note: the changed_cb will be called once before the register
         * func returns, thus changing the checksum/compression from the
         * default (fletcher2/off).  Snapshots don't need to know about
         * checksum/compression/copies.
         */
        if (ds != NULL) {
                err = dsl_prop_register(ds,
                    zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
                    primary_cache_changed_cb, os);
                if (err == 0) {
                        err = dsl_prop_register(ds,
                            zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
                            secondary_cache_changed_cb, os);
                }
                if (!ds->ds_is_snapshot) {
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(ZFS_PROP_CHECKSUM),
                                    checksum_changed_cb, os);
                        }
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(ZFS_PROP_COMPRESSION),
                                    compression_changed_cb, os);
                        }
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(ZFS_PROP_COPIES),
                                    copies_changed_cb, os);
                        }
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(ZFS_PROP_DEDUP),
                                    dedup_changed_cb, os);
                        }
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(ZFS_PROP_LOGBIAS),
                                    logbias_changed_cb, os);
                        }
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(ZFS_PROP_SYNC),
                                    sync_changed_cb, os);
                        }
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(
                                    ZFS_PROP_REDUNDANT_METADATA),
                                    redundant_metadata_changed_cb, os);
                        }
                        if (err == 0) {
                                err = dsl_prop_register(ds,
                                    zfs_prop_to_name(ZFS_PROP_RECORDSIZE),
                                    recordsize_changed_cb, os);
                        }
                }
                if (err != 0) {
                        VERIFY(arc_buf_remove_ref(os->os_phys_buf,
                            &os->os_phys_buf));
                        kmem_free(os, sizeof (objset_t));
                        return (err);
                }
        } else {
                /* It’s the meta-objset. */
                os->os_checksum = ZIO_CHECKSUM_FLETCHER_4;
                os->os_compress = ZIO_COMPRESS_ON;
                os->os_copies = spa_max_replication(spa);
                os->os_dedup_checksum = ZIO_CHECKSUM_OFF;
                os->os_dedup_verify = B_FALSE;
                os->os_logbias = ZFS_LOGBIAS_LATENCY;
                os->os_sync = ZFS_SYNC_STANDARD;
                os->os_primary_cache = ZFS_CACHE_ALL;
                os->os_secondary_cache = ZFS_CACHE_ALL;
        }

        if (ds == NULL || !ds->ds_is_snapshot)
                os->os_zil_header = os->os_phys->os_zil_header;
        os->os_zil = zil_alloc(os, &os->os_zil_header);

        for (i = 0; i < TXG_SIZE; i++) {
                list_create(&os->os_dirty_dnodes[i], sizeof (dnode_t),
                    offsetof(dnode_t, dn_dirty_link[i]));
                list_create(&os->os_free_dnodes[i], sizeof (dnode_t),
                    offsetof(dnode_t, dn_dirty_link[i]));
        }
```

```
        list_create(&os->os_dnodes, sizeof (dnode_t),
            offsetof(dnode_t, dn_link));
        list_create(&os->os_downgraded_dbufs, sizeof (dmu_buf_impl_t),
            offsetof(dmu_buf_impl_t, db_link));

        mutex_init(&os->os_lock, NULL, MUTEX_DEFAULT, NULL);
        mutex_init(&os->os_obj_lock, NULL, MUTEX_DEFAULT, NULL);
        mutex_init(&os->os_user_ptr_lock, NULL, MUTEX_DEFAULT, NULL);

        dnode_special_open(os, &os->os_phys->os_meta_dnode,
            DMU_META_DNODE_OBJECT, &os->os_meta_dnode);
        if (arc_buf_size(os->os_phys_buf) >= sizeof (objset_phys_t)) {
                dnode_special_open(os, &os->os_phys->os_userused_dnode,
                    DMU_USERUSED_OBJECT, &os->os_userused_dnode);
                dnode_special_open(os, &os->os_phys->os_groupused_dnode,
                    DMU_GROUPUSED_OBJECT, &os->os_groupused_dnode);
        }

        *osp = os;
        return (0);
}

int
dmu_objset_from_ds(dsl_dataset_t *ds, objset_t **osp)
{
        int err = 0;

        mutex_enter(&ds->ds_opening_lock);
        if (ds->ds_objset == NULL) {
                objset_t *os;
                err = dmu_objset_open_impl(dsl_dataset_get_spa(ds),
                    ds, dsl_dataset_get_blkptr(ds), &os);

                if (err == 0) {
                        mutex_enter(&ds->ds_lock);
                        ASSERT(ds->ds_objset == NULL);
                        ds->ds_objset = os;
                        mutex_exit(&ds->ds_lock);
                }
        }
        *osp = ds->ds_objset;
        mutex_exit(&ds->ds_opening_lock);
        return (err);
}

/*
 * Holds the pool while the objset is held.  Therefore only one objset
 * can be held at a time.
 */
int
dmu_objset_hold(const char *name, void *tag, objset_t **osp)
{
        dsl_pool_t *dp;
        dsl_dataset_t *ds;
        int err;

        err = dsl_pool_hold(name, tag, &dp);
        if (err != 0)
                return (err);
        err = dsl_dataset_hold(dp, name, tag, &ds);
        if (err != 0) {
                dsl_pool_rele(dp, tag);
                return (err);
        }

        err = dmu_objset_from_ds(ds, osp);
        if (err != 0) {
                dsl_dataset_rele(ds, tag);
                dsl_pool_rele(dp, tag);
        }

        return (err);
}

static int
dmu_objset_own_impl(dsl_dataset_t *ds, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp)
{
        int err;

        err = dmu_objset_from_ds(ds, osp);
        if (err != 0) {
                dsl_dataset_disown(ds, tag);
        } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
                dsl_dataset_disown(ds, tag);
                return (SET_ERROR(EINVAL));
        } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
                dsl_dataset_disown(ds, tag);
                return (SET_ERROR(EROFS));
        }
        return (err);
}

#endif /* ! codereview */
/*
 * dsl_pool must not be held when this is called.
 * Upon successful return, there will be a longhold on the dataset,
 * and the dsl_pool will not be held.
 */
int
dmu_objset_own(const char *name, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp)
{
        dsl_pool_t *dp;
        dsl_dataset_t *ds;
        int err;

        err = dsl_pool_hold(name, FTAG, &dp);
        if (err != 0)
                return (err);
        err = dsl_dataset_own(dp, name, tag, &ds);
        if (err != 0) {
                dsl_pool_rele(dp, FTAG);
                return (err);
        }
        err = dmu_objset_own_impl(ds, type, readonly, tag, osp);
        dsl_pool_rele(dp, FTAG);

        return (err);
}
#endif /* ! codereview */

int
dmu_objset_own_obj(dsl_pool_t *dp, uint64_t obj, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp)
{
        dsl_dataset_t *ds;
        int err;

        err = dsl_dataset_own_obj(dp, obj, tag, &ds);
        if (err != 0)
        err = dmu_objset_from_ds(ds, osp);
```

```
        dsl_pool_rele(dp, FTAG);
        if (err != 0) {
                dsl_dataset_disown(ds, tag);
        } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
                dsl_dataset_disown(ds, tag);
                return (SET_ERROR(EINVAL));
        } else if (!readonly && ds->ds_is_snapshot) {
                dsl_dataset_disown(ds, tag);
                return (SET_ERROR(EROFS));
        }
                return (err);

        return (dmu_objset_own_impl(ds, type, readonly, tag, osp));
#endif /* ! codereview */
}

void
dmu_objset_rele(objset_t *os, void *tag)
{
        dsl_pool_t *dp = dmu_objset_pool(os);
        dsl_dataset_rele(os->os_dsl_dataset, tag);
        dsl_pool_rele(dp, tag);
}

/*
 * When we are called, os MUST refer to an objset associated with a dataset
 * that is owned by 'tag'; that is, is held and long held by 'tag' and ds_owner
 * == tag.  We will then release and reacquire ownership of the dataset while
 * holding the pool config_rwlock to avoid intervening namespace or ownership
 * changes may occur.
 *
 * This exists solely to accommodate zfs_ioc_userspace_upgrade()'s desire to
 * release the hold on its dataset and acquire a new one on the dataset of the
 * same name so that it can be partially torn down and reconstructed.
 */
void
dmu_objset_refresh_ownership(objset_t *os, void *tag)
{
        dsl_pool_t *dp;
        dsl_dataset_t *ds, *newds;
        char name[MAXNAMELEN];

        ds = os->os_dsl_dataset;
        VERIFY3P(ds, !=, NULL);
        VERIFY3P(ds->ds_owner, ==, tag);
        VERIFY(dsl_dataset_long_held(ds));

        dsl_dataset_name(ds, name);
        dp = dmu_objset_pool(os);
        dsl_pool_config_enter(dp, FTAG);
        dmu_objset_disown(os, tag);
        VERIFY0(dsl_dataset_own(dp, name, tag, &newds));
        VERIFY3P(newds, ==, os->os_dsl_dataset);
        dsl_pool_config_exit(dp, FTAG);
}

void
dmu_objset_disown(objset_t *os, void *tag)
{
        dsl_dataset_disown(os->os_dsl_dataset, tag);
}

void
dmu_objset_evict_dbufs(objset_t *os)
{
        dnode_t dn_marker;
        dnode_t *dn;

        mutex_enter(&os->os_lock);
        dn = list_head(&os->os_dnodes);
        while (dn != NULL) {
                /*
                 * Skip dnodes without holds.  We have to do this dance
                 * because dnode_add_ref() only works if there is already a
                 * hold.  If the dnode has no holds, then it has no dbufs.
                 */
                if (dnode_add_ref(dn, FTAG)) {
                        list_insert_after(&os->os_dnodes, dn, &dn_marker);
                        mutex_exit(&os->os_lock);

                        dnode_evict_dbufs(dn);
                        dnode_rele(dn, FTAG);

                        mutex_enter(&os->os_lock);
                        dn = list_next(&os->os_dnodes, &dn_marker);
                        list_remove(&os->os_dnodes, &dn_marker);
                } else {
                        dn = list_next(&os->os_dnodes, dn);
                }
        }
        mutex_exit(&os->os_lock);

        if (DMU_USERUSED_DNODE(os) != NULL) {
                dnode_evict_dbufs(DMU_GROUPUSED_DNODE(os));
                dnode_evict_dbufs(DMU_USERUSED_DNODE(os));
        }
        dnode_evict_dbufs(DMU_META_DNODE(os));
}

/*
 * Objset eviction processing is split into into two pieces.
 * The first marks the objset as evicting, evicts any dbufs that
 * have a refcount of zero, and then queues up the objset for the
 * second phase of eviction.  Once os->os_dnodes has been cleared by
 * dnode_buf_pageout()->dnode_destroy(), the second phase is executed.
 * The second phase closes the special dnodes, dequeues the objset from
 * the list of those undergoing eviction, and finally frees the objset.
 *
 * NOTE: Due to asynchronous eviction processing (invocation of
 *       dnode_buf_pageout()), it is possible for the meta dnode for the
 *       objset to have no holds even though os->os_dnodes is not empty.
 */
void
dmu_objset_evict(objset_t *os)
{
        dsl_dataset_t *ds = os->os_dsl_dataset;

        for (int t = 0; t < TXG_SIZE; t++)
                ASSERT(!dmu_objset_is_dirty(os, t));

        if (ds) {
                if (!ds->ds_is_snapshot) {
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_CHECKSUM),
                            checksum_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_COMPRESSION),
                            compression_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_COPIES),
                            copies_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
```

```
                            zfs_prop_to_name(ZFS_PROP_DEDUP),
                            dedup_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_LOGBIAS),
                            logbias_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_SYNC),
                            sync_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_REDUNDANT_METADATA),
                            redundant_metadata_changed_cb, os));
                        VERIFY0(dsl_prop_unregister(ds,
                            zfs_prop_to_name(ZFS_PROP_RECORDSIZE),
                            recordsize_changed_cb, os));
                }
                VERIFY0(dsl_prop_unregister(ds,
                    zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
                    primary_cache_changed_cb, os));
                VERIFY0(dsl_prop_unregister(ds,
                    zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
                    secondary_cache_changed_cb, os));
        }

        if (os->os_sa)
                sa_tear_down(os);

        os->os_evicting = B_TRUE;
        dmu_objset_evict_dbufs(os);

        mutex_enter(&os->os_lock);
        spa_evicting_os_register(os->os_spa, os);
        if (list_is_empty(&os->os_dnodes)) {
                mutex_exit(&os->os_lock);
                dmu_objset_evict_done(os);
        } else {
                mutex_exit(&os->os_lock);
        }
}

void
dmu_objset_evict_done(objset_t *os)
{
        ASSERT3P(list_head(&os->os_dnodes), ==, NULL);

        dnode_special_close(&os->os_meta_dnode);
        if (DMU_USERUSED_DNODE(os)) {
                dnode_special_close(&os->os_userused_dnode);
                dnode_special_close(&os->os_groupused_dnode);
        }
        zil_free(os->os_zil);

        VERIFY(arc_buf_remove_ref(os->os_phys_buf, &os->os_phys_buf));

        /*
         * This is a barrier to prevent the objset from going away in
         * dnode_move() until we can safely ensure that the objset is still in
         * use. We consider the objset valid before the barrier and invalid
         * after the barrier.
         */
        rw_enter(&os_lock, RW_READER);
        rw_exit(&os_lock);

        mutex_destroy(&os->os_lock);
        mutex_destroy(&os->os_obj_lock);
        mutex_destroy(&os->os_user_ptr_lock);
        spa_evicting_os_deregister(os->os_spa, os);
```

```c
        kmem_free(os, sizeof (objset_t));
}

timestruc_t
dmu_objset_snap_cmtime(objset_t *os)
{
        return (dsl_dir_snap_cmtime(os->os_dsl_dataset->ds_dir));
}

/* called from dsl for meta-objset */
objset_t *
dmu_objset_create_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
    dmu_objset_type_t type, dmu_tx_t *tx)
{
        objset_t *os;
        dnode_t *mdn;

        ASSERT(dmu_tx_is_syncing(tx));

        if (ds != NULL)
                VERIFY0(dmu_objset_from_ds(ds, &os));
        else
                VERIFY0(dmu_objset_open_impl(spa, NULL, bp, &os));

        mdn = DMU_META_DNODE(os);

        dnode_allocate(mdn, DMU_OT_DNODE, 1 << DNODE_BLOCK_SHIFT,
            DN_MAX_INDBLKSHIFT, DMU_OT_NONE, 0, tx);

        /*
         * We don’t want to have to increase the meta-dnode’s nlevels
         * later, because then we could do it in quescing context while
         * we are also accessing it in open context.
         *
         * This precaution is not necessary for the MOS (ds == NULL),
         * because the MOS is only updated in syncing context.
         * This is most fortunate: the MOS is the only objset that
         * needs to be synced multiple times as spa_sync() iterates
         * to convergence, so minimizing its dn_nlevels matters.
         */
        if (ds != NULL) {
                int levels = 1;

                /*
                 * Determine the number of levels necessary for the meta-dnode
                 * to contain DN_MAX_OBJECT dnodes.
                 */
                while ((uint64_t)mdn->dn_nblkptr << (mdn->dn_datablkshift +
                    (levels - 1) * (mdn->dn_indblkshift - SPA_BLKPTRSHIFT)) <
                    DN_MAX_OBJECT * sizeof (dnode_phys_t))
                        levels++;

                mdn->dn_next_nlevels[tx->tx_txg & TXG_MASK] =
                    mdn->dn_nlevels = levels;
        }

        ASSERT(type != DMU_OST_NONE);
        ASSERT(type != DMU_OST_ANY);
        ASSERT(type < DMU_OST_NUMTYPES);
        os->os_phys->os_type = type;
        if (dmu_objset_userused_enabled(os)) {
                os->os_phys->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
                os->os_flags = os->os_phys->os_flags;
        }

        dsl_dataset_dirty(ds, tx);
```

```c
        return (os);
}

typedef struct dmu_objset_create_arg {
        const char *doca_name;
        cred_t *doca_cred;
        void (*doca_userfunc)(objset_t *os, void *arg,
            cred_t *cr, dmu_tx_t *tx);
        void *doca_userarg;
        dmu_objset_type_t doca_type;
        uint64_t doca_flags;
} dmu_objset_create_arg_t;

/*ARGSUSED*/
static int
dmu_objset_create_check(void *arg, dmu_tx_t *tx)
{
        dmu_objset_create_arg_t *doca = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dir_t *pdd;
        const char *tail;
        int error;

        if (strchr(doca->doca_name, ’@’) != NULL)
                return (SET_ERROR(EINVAL));

        error = dsl_dir_hold(dp, doca->doca_name, FTAG, &pdd, &tail);
        if (error != 0)
                return (error);
        if (tail == NULL) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EEXIST));
        }
        error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
            doca->doca_cred);
        dsl_dir_rele(pdd, FTAG);

        return (error);
}

static void
dmu_objset_create_sync(void *arg, dmu_tx_t *tx)
{
        dmu_objset_create_arg_t *doca = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dir_t *pdd;
        const char *tail;
        dsl_dataset_t *ds;
        uint64_t obj;
        blkptr_t *bp;
        objset_t *os;

        VERIFY0(dsl_dir_hold(dp, doca->doca_name, FTAG, &pdd, &tail));

        obj = dsl_dataset_create_sync(pdd, tail, NULL, doca->doca_flags,
            doca->doca_cred, tx);

        VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
        bp = dsl_dataset_get_blkptr(ds);
        os = dmu_objset_create_impl(pdd->dd_pool->dp_spa,
            ds, bp, doca->doca_type, tx);

        if (doca->doca_userfunc != NULL) {
                doca->doca_userfunc(os, doca->doca_userarg,
                    doca->doca_cred, tx);
        }

        spa_history_log_internal_ds(ds, "create", tx, "");
        dsl_dataset_rele(ds, FTAG);
        dsl_dir_rele(pdd, FTAG);
}

int
dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
    void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg)
{
        dmu_objset_create_arg_t doca;

        doca.doca_name = name;
        doca.doca_cred = CRED();
        doca.doca_flags = flags;
        doca.doca_userfunc = func;
        doca.doca_userarg = arg;
        doca.doca_type = type;

        return (dsl_sync_task(name,
            dmu_objset_create_check, dmu_objset_create_sync, &doca,
            5, ZFS_SPACE_CHECK_NORMAL));
}

typedef struct dmu_objset_clone_arg {
        const char *doca_clone;
        const char *doca_origin;
        cred_t *doca_cred;
} dmu_objset_clone_arg_t;

/*ARGSUSED*/
static int
dmu_objset_clone_check(void *arg, dmu_tx_t *tx)
{
        dmu_objset_clone_arg_t *doca = arg;
        dsl_dir_t *pdd;
        const char *tail;
        int error;
        dsl_dataset_t *origin;
        dsl_pool_t *dp = dmu_tx_pool(tx);

        if (strchr(doca->doca_clone, ’@’) != NULL)
                return (SET_ERROR(EINVAL));

        error = dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail);
        if (error != 0)
                return (error);
        if (tail == NULL) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EEXIST));
        }

        error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
            doca->doca_cred);
        if (error != 0) {
                dsl_dir_rele(pdd, FTAG);
                return (SET_ERROR(EDQUOT));
        }
        dsl_dir_rele(pdd, FTAG);

        error = dsl_dataset_hold(dp, doca->doca_origin, FTAG, &origin);
        if (error != 0)
                return (error);

        /* You can only clone snapshots, not the head datasets. */
```

```
        if (!origin->ds_is_snapshot) {
                dsl_dataset_rele(origin, FTAG);
                return (SET_ERROR(EINVAL));
        }
        dsl_dataset_rele(origin, FTAG);

        return (0);
}

static void
dmu_objset_clone_sync(void *arg, dmu_tx_t *tx)
{
        dmu_objset_clone_arg_t *doca = arg;
        dsl_pool_t *dp = dmu_tx_pool(tx);
        dsl_dir_t *pdd;
        const char *tail;
        dsl_dataset_t *origin, *ds;
        uint64_t obj;
        char namebuf[MAXNAMELEN];

        VERIFY0(dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail));
        VERIFY0(dsl_dataset_hold(dp, doca->doca_origin, FTAG, &origin));

        obj = dsl_dataset_create_sync(pdd, tail, origin, 0,
            doca->doca_cred, tx);

        VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
        dsl_dataset_name(origin, namebuf);
        spa_history_log_internal_ds(ds, "clone", tx,
            "origin=%s (%llu)", namebuf, origin->ds_object);
        dsl_dataset_rele(ds, FTAG);
        dsl_dataset_rele(origin, FTAG);
        dsl_dir_rele(pdd, FTAG);
}

int
dmu_objset_clone(const char *clone, const char *origin)
{
        dmu_objset_clone_arg_t doca;

        doca.doca_clone = clone;
        doca.doca_origin = origin;
        doca.doca_cred = CRED();

        return (dsl_sync_task(clone,
            dmu_objset_clone_check, dmu_objset_clone_sync, &doca,
            5, ZFS_SPACE_CHECK_NORMAL));
}

int
dmu_objset_snapshot_one(const char *fsname, const char *snapname)
{
        int err;
        char *longsnap = kmem_asprintf("%s@%s", fsname, snapname);
        nvlist_t *snaps = fnvlist_alloc();

        fnvlist_add_boolean(snaps, longsnap);
        strfree(longsnap);
        err = dsl_dataset_snapshot(snaps, NULL, NULL);
        fnvlist_free(snaps);
        return (err);
}

static void
dmu_objset_sync_dnodes(list_t *list, list_t *newlist, dmu_tx_t *tx)
{
        dnode_t *dn;

        while (dn = list_head(list)) {
                ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
                ASSERT(dn->dn_dbuf->db_data_pending);
                /*
                 * Initialize dn_zio outside dnode_sync() because the
                 * meta-dnode needs to set it ouside dnode_sync().
                 */
                dn->dn_zio = dn->dn_dbuf->db_data_pending->dr_zio;
                ASSERT(dn->dn_zio);

                ASSERT3U(dn->dn_nlevels, <=, DN_MAX_LEVELS);
                list_remove(list, dn);

                if (newlist) {
                        (void) dnode_add_ref(dn, newlist);
                        list_insert_tail(newlist, dn);
                }

                dnode_sync(dn, tx);
        }
}

/* ARGSUSED */
static void
dmu_objset_write_ready(zio_t *zio, arc_buf_t *abuf, void *arg)
{
        blkptr_t *bp = zio->io_bp;
        objset_t *os = arg;
        dnode_phys_t *dnp = &os->os_phys->os_meta_dnode;

        ASSERT(!BP_IS_EMBEDDED(bp));
        ASSERT3P(bp, ==, os->os_rootbp);
        ASSERT3U(BP_GET_TYPE(bp), ==, DMU_OT_OBJSET);
        ASSERT0(BP_GET_LEVEL(bp));

        /*
         * Update rootbp fill count: it should be the number of objects
         * allocated in the object set (not counting the "special"
         * objects that are stored in the objset_phys_t -- the meta
         * dnode and user/group accounting objects).
         */
        bp->blk_fill = 0;
        for (int i = 0; i < dnp->dn_nblkptr; i++)
                bp->blk_fill += BP_GET_FILL(&dnp->dn_blkptr[i]);
}

/* ARGSUSED */
static void
dmu_objset_write_done(zio_t *zio, arc_buf_t *abuf, void *arg)
{
        blkptr_t *bp = zio->io_bp;
        blkptr_t *bp_orig = &zio->io_bp_orig;
        objset_t *os = arg;

        if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
                ASSERT(BP_EQUAL(bp, bp_orig));
        } else {
                dsl_dataset_t *ds = os->os_dsl_dataset;
                dmu_tx_t *tx = os->os_synctx;

                (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
                dsl_dataset_block_born(ds, bp, tx);
        }
}
```

```
/* called from dsl */
void
dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
{
        int txgoff;
        zbookmark_phys_t zb;
        zio_prop_t zp;
        zio_t *zio;
        list_t *list;
        list_t *newlist = NULL;
        dbuf_dirty_record_t *dr;

        dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);

        ASSERT(dmu_tx_is_syncing(tx));
        /* XXX the write_done callback should really give us the tx... */
        os->os_synctx = tx;

        if (os->os_dsl_dataset == NULL) {
                /*
                 * This is the MOS.  If we have upgraded,
                 * spa_max_replication() could change, so reset
                 * os_copies here.
                 */
                os->os_copies = spa_max_replication(os->os_spa);
        }

        /*
         * Create the root block IO
         */
        SET_BOOKMARK(&zb, os->os_dsl_dataset ?
            os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
            ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
        arc_release(os->os_phys_buf, &os->os_phys_buf);

        dmu_write_policy(os, NULL, 0, 0, &zp);

        zio = arc_write(pio, os->os_spa, tx->tx_txg,
            os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os),
            DMU_OS_IS_L2COMPRESSIBLE(os), &zp, dmu_objset_write_ready,
            NULL, dmu_objset_write_done, os, ZIO_PRIORITY_ASYNC_WRITE,
            ZIO_FLAG_MUSTSUCCEED, &zb);

        /*
         * Sync special dnodes - the parent IO for the sync is the root block
         */
        DMU_META_DNODE(os)->dn_zio = zio;
        dnode_sync(DMU_META_DNODE(os), tx);

        os->os_phys->os_flags = os->os_flags;

        if (DMU_USERUSED_DNODE(os) &&
            DMU_USERUSED_DNODE(os)->dn_type != DMU_OT_NONE) {
                DMU_USERUSED_DNODE(os)->dn_zio = zio;
                dnode_sync(DMU_USERUSED_DNODE(os), tx);
                DMU_GROUPUSED_DNODE(os)->dn_zio = zio;
                dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
        }

        txgoff = tx->tx_txg & TXG_MASK;

        if (dmu_objset_userused_enabled(os)) {
                newlist = &os->os_synced_dnodes;
                /*
                 * We must create the list here because it uses the
                 * dn_dirty_link[] of this txg.
                 */
                list_create(newlist, sizeof (dnode_t),
                    offsetof(dnode_t, dn_dirty_link[txgoff]));
        }

        dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
        dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);

        list = &DMU_META_DNODE(os)->dn_dirty_records[txgoff];
        while (dr = list_head(list)) {
                ASSERT0(dr->dr_dbuf->db_level);
                list_remove(list, dr);
                if (dr->dr_zio)
                        zio_nowait(dr->dr_zio);
        }
        /*
         * Free intent log blocks up to this tx.
         */
        zil_sync(os->os_zil, tx);
        os->os_phys->os_zil_header = os->os_zil_header;
        zio_nowait(zio);
}

boolean_t
dmu_objset_is_dirty(objset_t *os, uint64_t txg)
{
        return (!list_is_empty(&os->os_dirty_dnodes[txg & TXG_MASK]) ||
            !list_is_empty(&os->os_free_dnodes[txg & TXG_MASK]));
}

static objset_used_cb_t *used_cbs[DMU_OST_NUMTYPES];

void
dmu_objset_register_type(dmu_objset_type_t ost, objset_used_cb_t *cb)
{
        used_cbs[ost] = cb;
}

boolean_t
dmu_objset_userused_enabled(objset_t *os)
{
        return (spa_version(os->os_spa) >= SPA_VERSION_USERSPACE &&
            used_cbs[os->os_phys->os_type] != NULL &&
            DMU_USERUSED_DNODE(os) != NULL);
}

static void
do_userquota_update(objset_t *os, uint64_t used, uint64_t flags,
    uint64_t user, uint64_t group, boolean_t subtract, dmu_tx_t *tx)
{
        if ((flags & DNODE_FLAG_USERUSED_ACCOUNTED)) {
                int64_t delta = DNODE_SIZE + used;
                if (subtract)
                        delta = -delta;
                VERIFY3U(0, ==, zap_increment_int(os, DMU_USERUSED_OBJECT,
                    user, delta, tx));
                VERIFY3U(0, ==, zap_increment_int(os, DMU_GROUPUSED_OBJECT,
                    group, delta, tx));
        }
}

void
dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx)
{
        dnode_t *dn;
```

```
        list_t *list = &os->os_synced_dnodes;

        ASSERT(list_head(list) == NULL || dmu_objset_userused_enabled(os));

        while (dn = list_head(list)) {
                int flags;
                ASSERT(!DMU_OBJECT_IS_SPECIAL(dn->dn_object));
                ASSERT(dn->dn_phys->dn_type == DMU_OT_NONE ||
                    dn->dn_phys->dn_flags &
                    DNODE_FLAG_USERUSED_ACCOUNTED);

                /* Allocate the user/groupused objects if necessary. */
                if (DMU_USERUSED_DNODE(os)->dn_type == DMU_OT_NONE) {
                        VERIFY(0 == zap_create_claim(os,
                            DMU_USERUSED_OBJECT,
                            DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
                        VERIFY(0 == zap_create_claim(os,
                            DMU_GROUPUSED_OBJECT,
                            DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
                }

                /*
                 * We intentionally modify the zap object even if the
                 * net delta is zero.  Otherwise
                 * the block of the zap obj could be shared between
                 * datasets but need to be different between them after
                 * a bprewrite.
                 */

                flags = dn->dn_id_flags;
                ASSERT(flags);
                if (flags & DN_ID_OLD_EXIST)  {
                        do_userquota_update(os, dn->dn_oldused, dn->dn_oldflags,
                            dn->dn_olduid, dn->dn_oldgid, B_TRUE, tx);
                }
                if (flags & DN_ID_NEW_EXIST) {
                        do_userquota_update(os, DN_USED_BYTES(dn->dn_phys),
                            dn->dn_phys->dn_flags,  dn->dn_newuid,
                            dn->dn_newgid, B_FALSE, tx);
                }

                mutex_enter(&dn->dn_mtx);
                dn->dn_oldused = 0;
                dn->dn_oldflags = 0;
                if (dn->dn_id_flags & DN_ID_NEW_EXIST) {
                        dn->dn_olduid = dn->dn_newuid;
                        dn->dn_oldgid = dn->dn_newgid;
                        dn->dn_id_flags |= DN_ID_OLD_EXIST;
                        if (dn->dn_bonuslen == 0)
                                dn->dn_id_flags |= DN_ID_CHKED_SPILL;
                        else
                                dn->dn_id_flags |= DN_ID_CHKED_BONUS;
                }
                dn->dn_id_flags &= ~(DN_ID_NEW_EXIST);
                mutex_exit(&dn->dn_mtx);

                list_remove(list, dn);
                dnode_rele(dn, list);
        }
}

/*
 * Returns a pointer to data to find uid/gid from
 *
 * If a dirty record for transaction group that is syncing can’t
 * be found then NULL is returned.  In the NULL case it is assumed
 * the uid/gid aren’t changing.
 */
static void *
dmu_objset_userquota_find_data(dmu_buf_impl_t *db, dmu_tx_t *tx)
{
        dbuf_dirty_record_t *dr, **drp;
        void *data;

        if (db->db_dirtycnt == 0)
                return (db->db.db_data);  /* Nothing is changing */

        for (drp = &db->db_last_dirty; (dr = *drp) != NULL; drp = &dr->dr_next)
                if (dr->dr_txg == tx->tx_txg)
                        break;

        if (dr == NULL) {
                data = NULL;
        } else {
                dnode_t *dn;

                DB_DNODE_ENTER(dr->dr_dbuf);
                dn = DB_DNODE(dr->dr_dbuf);

                if (dn->dn_bonuslen == 0 &&
                    dr->dr_dbuf->db_blkid == DMU_SPILL_BLKID)
                        data = dr->dt.dl.dr_data->b_data;
                else
                        data = dr->dt.dl.dr_data;

                DB_DNODE_EXIT(dr->dr_dbuf);
        }

        return (data);
}

void
dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx)
{
        objset_t *os = dn->dn_objset;
        void *data = NULL;
        dmu_buf_impl_t *db = NULL;
        uint64_t *user = NULL;
        uint64_t *group = NULL;
        int flags = dn->dn_id_flags;
        int error;
        boolean_t have_spill = B_FALSE;

        if (!dmu_objset_userused_enabled(dn->dn_objset))
                return;

        if (before && (flags & (DN_ID_CHKED_BONUS|DN_ID_OLD_EXIST|
            DN_ID_CHKED_SPILL)))
                return;

        if (before && dn->dn_bonuslen != 0)
                data = DN_BONUS(dn->dn_phys);
        else if (!before && dn->dn_bonuslen != 0) {
                if (dn->dn_bonus) {
                        db = dn->dn_bonus;
                        mutex_enter(&db->db_mtx);
                        data = dmu_objset_userquota_find_data(db, tx);
                } else {
                        data = DN_BONUS(dn->dn_phys);
                }
        } else if (dn->dn_bonuslen == 0 && dn->dn_bonustype == DMU_OT_SA) {
                        int rf = 0;
```

```
                        if (RW_WRITE_HELD(&dn->dn_struct_rwlock))
                                rf |= DB_RF_HAVESTRUCT;
                        error = dmu_spill_hold_by_dnode(dn,
                            rf | DB_RF_MUST_SUCCEED,
                            FTAG, (dmu_buf_t **)&db);
                        ASSERT(error == 0);
                        mutex_enter(&db->db_mtx);
                        data = (before) ? db->db.db_data :
                            dmu_objset_userquota_find_data(db, tx);
                        have_spill = B_TRUE;
        } else {
                mutex_enter(&dn->dn_mtx);
                dn->dn_id_flags |= DN_ID_CHKED_BONUS;
                mutex_exit(&dn->dn_mtx);
                return;
        }

        if (before) {
                ASSERT(data);
                user = &dn->dn_olduid;
                group = &dn->dn_oldgid;
        } else if (data) {
                user = &dn->dn_newuid;
                group = &dn->dn_newgid;
        }

        /*
         * Must always call the callback in case the object
         * type has changed and that type isn’t an object type to track
         */
        error = used_cbs[os->os_phys->os_type](dn->dn_bonustype, data,
            user, group);

        /*
         * Preserve existing uid/gid when the callback can’t determine
         * what the new uid/gid are and the callback returned EEXIST.
         * The EEXIST error tells us to just use the existing uid/gid.
         * If we don’t know what the old values are then just assign
         * them to 0, since that is a new file  being created.
         */
        if (!before && data == NULL && error == EEXIST) {
                if (flags & DN_ID_OLD_EXIST) {
                        dn->dn_newuid = dn->dn_olduid;
                        dn->dn_newgid = dn->dn_oldgid;
                } else {
                        dn->dn_newuid = 0;
                        dn->dn_newgid = 0;
                }
                error = 0;
        }

        if (db)
                mutex_exit(&db->db_mtx);

        mutex_enter(&dn->dn_mtx);
        if (error == 0 && before)
                dn->dn_id_flags |= DN_ID_OLD_EXIST;
        if (error == 0 && !before)
                dn->dn_id_flags |= DN_ID_NEW_EXIST;

        if (have_spill) {
                dn->dn_id_flags |= DN_ID_CHKED_SPILL;
        } else {
                dn->dn_id_flags |= DN_ID_CHKED_BONUS;
        }
        mutex_exit(&dn->dn_mtx);
        if (have_spill)
                dmu_buf_rele((dmu_buf_t *)db, FTAG);
}

boolean_t
dmu_objset_userspace_present(objset_t *os)
{
        return (os->os_phys->os_flags &
            OBJSET_FLAG_USERACCOUNTING_COMPLETE);
}

int
dmu_objset_userspace_upgrade(objset_t *os)
{
        uint64_t obj;
        int err = 0;

        if (dmu_objset_userspace_present(os))
                return (0);
        if (!dmu_objset_userused_enabled(os))
                return (SET_ERROR(ENOTSUP));
        if (dmu_objset_is_snapshot(os))
                return (SET_ERROR(EINVAL));

        /*
         * We simply need to mark every object dirty, so that it will be
         * synced out and now accounted.  If this is called
         * concurrently, or if we already did some work before crashing,
         * that's fine, since we track each object's accounted state
         * independently.
         */

        for (obj = 0; err == 0; err = dmu_object_next(os, &obj, FALSE, 0)) {
                dmu_tx_t *tx;
                dmu_buf_t *db;
                int objerr;

                if (issig(JUSTLOOKING) && issig(FORREAL))
                        return (SET_ERROR(EINTR));

                objerr = dmu_bonus_hold(os, obj, FTAG, &db);
                if (objerr != 0)
                        continue;
                tx = dmu_tx_create(os);
                dmu_tx_hold_bonus(tx, obj);
                objerr = dmu_tx_assign(tx, TXG_WAIT);
                if (objerr != 0) {
                        dmu_tx_abort(tx);
                        continue;
                }
                dmu_buf_will_dirty(db, tx);
                dmu_buf_rele(db, FTAG);
                dmu_tx_commit(tx);
        }

        os->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
        txg_wait_synced(dmu_objset_pool(os), 0);
        return (0);
}

void
dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp)
{
        dsl_dataset_space(os->os_dsl_dataset, refdbytesp, availbytesp,
```

```
            usedobjsp, availobjsp);
}

uint64_t
dmu_objset_fsid_guid(objset_t *os)
{
        return (dsl_dataset_fsid_guid(os->os_dsl_dataset));
}

void
dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat)
{
        stat->dds_type = os->os_phys->os_type;
        if (os->os_dsl_dataset)
                dsl_dataset_fast_stat(os->os_dsl_dataset, stat);
}

void
dmu_objset_stats(objset_t *os, nvlist_t *nv)
{
        ASSERT(os->os_dsl_dataset ||
            os->os_phys->os_type == DMU_OST_META);

        if (os->os_dsl_dataset != NULL)
                dsl_dataset_stats(os->os_dsl_dataset, nv);

        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_TYPE,
            os->os_phys->os_type);
        dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERACCOUNTING,
            dmu_objset_userspace_present(os));
}

int
dmu_objset_is_snapshot(objset_t *os)
{
        if (os->os_dsl_dataset != NULL)
                return (os->os_dsl_dataset->ds_is_snapshot);
        else
                return (B_FALSE);
}

int
dmu_snapshot_realname(objset_t *os, char *name, char *real, int maxlen,
    boolean_t *conflict)
{
        dsl_dataset_t *ds = os->os_dsl_dataset;
        uint64_t ignored;

        if (dsl_dataset_phys(ds)->ds_snapnames_zapobj == 0)
                return (SET_ERROR(ENOENT));

        return (zap_lookup_norm(ds->ds_dir->dd_pool->dp_meta_objset,
            dsl_dataset_phys(ds)->ds_snapnames_zapobj, name, 8, 1, &ignored,
            MT_FIRST, real, maxlen, conflict));
}

int
dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
    uint64_t *idp, uint64_t *offp, boolean_t *case_conflict)
{
        dsl_dataset_t *ds = os->os_dsl_dataset;
        zap_cursor_t cursor;
        zap_attribute_t attr;

        ASSERT(dsl_pool_config_held(dmu_objset_pool(os)));
        if (dsl_dataset_phys(ds)->ds_snapnames_zapobj == 0)
                return (SET_ERROR(ENOENT));

        zap_cursor_init_serialized(&cursor,
            ds->ds_dir->dd_pool->dp_meta_objset,
            dsl_dataset_phys(ds)->ds_snapnames_zapobj, *offp);

        if (zap_cursor_retrieve(&cursor, &attr) != 0) {
                zap_cursor_fini(&cursor);
                return (SET_ERROR(ENOENT));
        }

        if (strlen(attr.za_name) + 1 > namelen) {
                zap_cursor_fini(&cursor);
                return (SET_ERROR(ENAMETOOLONG));
        }

        (void) strcpy(name, attr.za_name);
        if (idp)
                *idp = attr.za_first_integer;
        if (case_conflict)
                *case_conflict = attr.za_normalization_conflict;
        zap_cursor_advance(&cursor);
        *offp = zap_cursor_serialize(&cursor);
        zap_cursor_fini(&cursor);

        return (0);
}

int
dmu_dir_list_next(objset_t *os, int namelen, char *name,
    uint64_t *idp, uint64_t *offp)
{
        dsl_dir_t *dd = os->os_dsl_dataset->ds_dir;
        zap_cursor_t cursor;
        zap_attribute_t attr;

        /* there is no next dir on a snapshot! */
        if (os->os_dsl_dataset->ds_object !=
            dsl_dir_phys(dd)->dd_head_dataset_obj)
                return (SET_ERROR(ENOENT));

        zap_cursor_init_serialized(&cursor,
            dd->dd_pool->dp_meta_objset,
            dsl_dir_phys(dd)->dd_child_dir_zapobj, *offp);

        if (zap_cursor_retrieve(&cursor, &attr) != 0) {
                zap_cursor_fini(&cursor);
                return (SET_ERROR(ENOENT));
        }

        if (strlen(attr.za_name) + 1 > namelen) {
                zap_cursor_fini(&cursor);
                return (SET_ERROR(ENAMETOOLONG));
        }

        (void) strcpy(name, attr.za_name);
        if (idp)
                *idp = attr.za_first_integer;
        zap_cursor_advance(&cursor);
        *offp = zap_cursor_serialize(&cursor);
        zap_cursor_fini(&cursor);

        return (0);
}
```

```
typedef struct dmu_objset_find_ctx {
        taskq_t         *dc_tq;
        dsl_pool_t      *dc_dp;
        uint64_t        dc_ddobj;
        int             (*dc_func)(dsl_pool_t *, dsl_dataset_t *, void *);
        void            *dc_arg;
        int             dc_flags;
        kmutex_t        *dc_error_lock;
        int             *dc_error;
} dmu_objset_find_ctx_t;

static void
dmu_objset_find_dp_impl(dmu_objset_find_ctx_t *dcp)
/*
 * Find objsets under and including ddobj, call func(ds) on each.
 */
int
dmu_objset_find_dp(dsl_pool_t *dp, uint64_t ddobj,
    int func(dsl_pool_t *, dsl_dataset_t *, void *), void *arg, int flags)
{
        dsl_pool_t *dp = dcp->dc_dp;
        dmu_objset_find_ctx_t *child_dcp;
#endif /* ! codereview */
        dsl_dir_t *dd;
        dsl_dataset_t *ds;
        zap_cursor_t zc;
        zap_attribute_t *attr;
        uint64_t thisobj;
        int err = 0;
        int err;

        /* don't process if there already was an error */
        if (*dcp->dc_error != 0)
                goto out;
        ASSERT(dsl_pool_config_held(dp));

        err = dsl_dir_hold_obj(dp, dcp->dc_ddobj, NULL, FTAG, &dd);
        err = dsl_dir_hold_obj(dp, ddobj, NULL, FTAG, &dd);
        if (err != 0)
                goto out;
                return (err);

        /* Don't visit hidden ($MOS & $ORIGIN) objsets. */
        if (dd->dd_myname[0] == '$') {
                dsl_dir_rele(dd, FTAG);
                goto out;
                return (0);
        }

        thisobj = dsl_dir_phys(dd)->dd_head_dataset_obj;
        attr = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);

        /*
         * Iterate over all children.
         */
        if (dcp->dc_flags & DS_FIND_CHILDREN) {
        if (flags & DS_FIND_CHILDREN) {
                for (zap_cursor_init(&zc, dp->dp_meta_objset,
                    dsl_dir_phys(dd)->dd_child_dir_zapobj);
                    zap_cursor_retrieve(&zc, attr) == 0;
                    (void) zap_cursor_advance(&zc)) {
                        ASSERT3U(attr->za_integer_length, ==,
                            sizeof (uint64_t));
                        ASSERT3U(attr->za_num_integers, ==, 1);

                        child_dcp = kmem_alloc(sizeof(*child_dcp), KM_SLEEP);
                        *child_dcp = *dcp;
                        child_dcp->dc_ddobj = attr->za_first_integer;
                        if (dcp->dc_tq != NULL)
                                (void) taskq_dispatch(dcp->dc_tq,
                                    dmu_objset_find_dp_cb, child_dcp, TQ_SLEEP);
                        else
                                dmu_objset_find_dp_impl(child_dcp);
                        err = dmu_objset_find_dp(dp, attr->za_first_integer,
                            func, arg, flags);
                        if (err != 0)
                                break;
                }
                zap_cursor_fini(&zc);

                if (err != 0) {
                        dsl_dir_rele(dd, FTAG);
                        kmem_free(attr, sizeof (zap_attribute_t));
                        return (err);
                }
        }

        /*
         * Iterate over all snapshots.
         */
        if (dcp->dc_flags & DS_FIND_SNAPSHOTS) {
        if (flags & DS_FIND_SNAPSHOTS) {
                dsl_dataset_t *ds;
                err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);

                if (err == 0) {
                        uint64_t snapobj;

                        snapobj = dsl_dataset_phys(ds)->ds_snapnames_zapobj;
                        dsl_dataset_rele(ds, FTAG);

                        for (zap_cursor_init(&zc, dp->dp_meta_objset, snapobj);
                            zap_cursor_retrieve(&zc, attr) == 0;
                            (void) zap_cursor_advance(&zc)) {
                                ASSERT3U(attr->za_integer_length, ==,
                                    sizeof (uint64_t));
                                ASSERT3U(attr->za_num_integers, ==, 1);

                                err = dsl_dataset_hold_obj(dp,
                                    attr->za_first_integer, FTAG, &ds);
                                if (err != 0)
                                        break;
                                err = dcp->dc_func(dp, ds, dcp->dc_arg);
                                err = func(dp, ds, arg);
                                dsl_dataset_rele(ds, FTAG);
                                if (err != 0)
                                        break;
                        }
                        zap_cursor_fini(&zc);
                }
        }

        dsl_dir_rele(dd, FTAG);
        kmem_free(attr, sizeof (zap_attribute_t));

        if (err != 0)
                goto out;
                return (err);

        /*
         * Apply to self.
         */
```

```
        err = dsl_dataset_hold_obj(dp, thisobj, FTAG, &ds);
        if (err != 0)
                goto out;
        err = dcp->dc_func(dp, ds, dcp->dc_arg);
                return (err);
        err = func(dp, ds, arg);
        dsl_dataset_rele(ds, FTAG);

out:
        if (err != 0) {
                mutex_enter(dcp->dc_error_lock);
                /* only keep first error */
                if (*dcp->dc_error == 0)
                        *dcp->dc_error = err;
                mutex_exit(dcp->dc_error_lock);
        }

        kmem_free(dcp, sizeof(*dcp));
}

static void
dmu_objset_find_dp_cb(void *arg)
{
        dmu_objset_find_ctx_t *dcp = arg;
        dsl_pool_t *dp = dcp->dc_dp;

        dsl_pool_config_enter(dp, FTAG);

        dmu_objset_find_dp_impl(dcp);

        dsl_pool_config_exit(dp, FTAG);
}

/*
 * Find objsets under and including ddobj, call func(ds) on each.
 * The order for the enumeration is completely undefined.
 * func is called with dsl_pool_config held.
 */
int
dmu_objset_find_dp(dsl_pool_t *dp, uint64_t ddobj,
    int func(dsl_pool_t *, dsl_dataset_t *, void *), void *arg, int flags)
{
        int error = 0;
        taskq_t *tq = NULL;
        int ntasks;
        dmu_objset_find_ctx_t *dcp;
        kmutex_t err_lock;

        mutex_init(&err_lock, NULL, MUTEX_DEFAULT, NULL);
        dcp = kmem_alloc(sizeof(*dcp), KM_SLEEP);
        dcp->dc_tq = NULL;
        dcp->dc_dp = dp;
        dcp->dc_ddobj = ddobj;
        dcp->dc_func = func;
        dcp->dc_arg = arg;
        dcp->dc_flags = flags;
        dcp->dc_error_lock = &err_lock;
        dcp->dc_error = &error;

        if ((flags & DS_FIND_SERIALIZE) || dsl_pool_config_held_writer(dp)) {
                /*
                 * In case a write lock is held we can't make use of
                 * parallelism, as down the stack of the worker threads
                 * the lock is asserted via dsl_pool_config_held.
                 * In case of a read lock this is solved by getting a read
                 * lock in each worker thread, which isn't possible in case
```

```
		 * of a writer lock. So we fall back to the synchronous path
		 * here.
		 * In the future it might be possible to get some magic into
		 * dsl_pool_config_held in a way that it returns true for
		 * the worker threads so that a single lock held from this
		 * thread suffices. For now, stay single threaded.
		 */
		dmu_objset_find_dp_impl(dcp);

		return (error);
	}

	ntasks = dmu_find_threads;
	if (ntasks == 0)
		ntasks = vdev_count_leaves(dp->dp_spa) * 4;
	tq = taskq_create("dmu_objset_find", ntasks, minclsyspri, ntasks,
	    INT_MAX, 0);
	if (tq == NULL) {
		kmem_free(dcp, sizeof(*dcp));
		return (SET_ERROR(ENOMEM));
	}
	dcp->dc_tq = tq;

	/* dcp will be freed by task */
	(void) taskq_dispatch(tq, dmu_objset_find_dp_cb, dcp, TQ_SLEEP);

	/*
	 * PORTING: this code relies on the property of taskq_wait to wait
	 * until no more tasks are queued and no more tasks are active. As
	 * we always queue new tasks from within other tasks, task_wait
	 * reliably waits for the full recursion to finish, even though we
	 * enqueue new tasks after taskq_wait has been called.
	 * On platforms other than illumos, taskq_wait may not have this
	 * property.
	 */
	taskq_wait(tq);
	taskq_destroy(tq);
	mutex_destroy(&err_lock);

	return (error);
	return (err);
}
______unchanged_portion_omitted_
```

****
   31296 Wed May 6 08:47:27 2015
new/usr/src/uts/common/fs/zfs/dsl_pool.c
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****

______unchanged_portion_omitted_

```
void
dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx)
{
        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT(dp->dp_origin_snap != NULL);

        VERIFY0(dmu_objset_find_dp(dp, dp->dp_root_dir_obj, upgrade_clones_cb,
            tx, DS_FIND_CHILDREN | DS_FIND_SERIALIZE));
            tx, DS_FIND_CHILDREN));
}
```

______unchanged_portion_omitted_

```
void
dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx)
{
        ASSERT(dmu_tx_is_syncing(tx));
        uint64_t obj;

        (void) dsl_dir_create_sync(dp, dp->dp_root_dir, FREE_DIR_NAME, tx);
        VERIFY0(dsl_pool_open_special_dir(dp,
            FREE_DIR_NAME, &dp->dp_free_dir));

        /*
         * We can't use bpobj_alloc(), because spa_version() still
         * returns the old version, and we need a new-version bpobj with
         * subobj support.  So call dmu_object_alloc() directly.
         */
        obj = dmu_object_alloc(dp->dp_meta_objset, DMU_OT_BPOBJ,
            SPA_OLD_MAXBLOCKSIZE, DMU_OT_BPOBJ_HDR, sizeof (bpobj_phys_t), tx);
        VERIFY0(zap_add(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
            DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj, tx));
        VERIFY0(bpobj_open(&dp->dp_free_bpobj, dp->dp_meta_objset, obj));

        VERIFY0(dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
            upgrade_dir_clones_cb, tx, DS_FIND_CHILDREN | DS_FIND_SERIALIZE));
            upgrade_dir_clones_cb, tx, DS_FIND_CHILDREN));
}
```

______unchanged_portion_omitted_

```
boolean_t
dsl_pool_config_held_writer(dsl_pool_t *dp)
{
        return (RRW_WRITE_HELD(&dp->dp_config_rwlock));
}
#endif /* ! codereview */
```

****
180818 Wed May 6 08:47:27 2015
new/usr/src/uts/common/fs/zfs/spa.c
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait until no more tasks are queued and no more tasks are active. As we always queue new tasks from within other tasks, task_wait reliably waits for the full recursion to finish, even though we enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****

______unchanged_portion_omitted_

```
/*
 * Check for missing log devices
 */
static boolean_t
spa_check_logs(spa_t *spa)
{
        boolean_t rv = B_FALSE;
        dsl_pool_t *dp = spa_get_dsl(spa);
#endif /* ! codereview */

        switch (spa->spa_log_state) {
        case SPA_LOG_MISSING:
                /* need to recheck in case slog has been restored */
        case SPA_LOG_UNKNOWN:
                rv = (dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
                    zil_check_log_chain, NULL, DS_FIND_CHILDREN) != 0);
                rv = (dmu_objset_find(spa->spa_name, zil_check_log_chain,
                    NULL, DS_FIND_CHILDREN) != 0);
                if (rv)
                        spa_set_log_state(spa, SPA_LOG_MISSING);
                break;
        }
        return (rv);
}
```

______unchanged_portion_omitted_

```
/*
 * Load an existing storage pool, using the pool's builtin spa_config as a
 * source of configuration information.
 */
static int
spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
    spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
    char **ereport)
{
        int error = 0;
        nvlist_t *nvroot = NULL;
        nvlist_t *label;
        vdev_t *rvd;
        uberblock_t *ub = &spa->spa_uberblock;
        uint64_t children, config_cache_txg = spa->spa_config_txg;
        int orig_mode = spa->spa_mode;
        int parse;
        uint64_t obj;
        boolean_t missing_feat_write = B_FALSE;

        /*
         * If this is an untrusted config, access the pool in read-only mode.
         * This prevents things like resilvering recently removed devices.
```

```
         */
        if (!mosconfig)
                spa->spa_mode = FREAD;

        ASSERT(MUTEX_HELD(&spa_namespace_lock));

        spa->spa_load_state = state;

        if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
                return (SET_ERROR(EINVAL));

        parse = (type == SPA_IMPORT_EXISTING ?
            VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);

        /*
         * Create "The Godfather" zio to hold all async IOs
         */
        spa->spa_async_zio_root = kmem_alloc(max_ncpus * sizeof (void *),
            KM_SLEEP);
        for (int i = 0; i < max_ncpus; i++) {
                spa->spa_async_zio_root[i] = zio_root(spa, NULL, NULL,
                    ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE |
                    ZIO_FLAG_GODFATHER);
        }

        /*
         * Parse the configuration into a vdev tree.  We explicitly set the
         * value that will be returned by spa_version() since parsing the
         * configuration requires knowing the version number.
         */
        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
        error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
        spa_config_exit(spa, SCL_ALL, FTAG);

        if (error != 0)
                return (error);

        ASSERT(spa->spa_root_vdev == rvd);

        if (type != SPA_IMPORT_ASSEMBLE) {
                ASSERT(spa_guid(spa) == pool_guid);
        }

        /*
         * Try to open all vdevs, loading each label in the process.
         */
        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
        error = vdev_open(rvd);
        spa_config_exit(spa, SCL_ALL, FTAG);
        if (error != 0)
                return (error);

        /*
         * We need to validate the vdev labels against the configuration that
         * we have in hand, which is dependent on the setting of mosconfig. If
         * mosconfig is true then we're validating the vdev labels based on
         * that config.  Otherwise, we're validating against the cached config
         * (zpool.cache) that was read when we loaded the zfs module, and then
         * later we will recursively call spa_load() and validate against
         * the vdev config.
         *
         * If we're assembling a new pool that's been split off from an
         * existing pool, the labels haven't yet been updated so we skip
         * validation for now.
         */
        if (type != SPA_IMPORT_ASSEMBLE) {
                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
                error = vdev_validate(rvd, mosconfig);
                spa_config_exit(spa, SCL_ALL, FTAG);

                if (error != 0)
                        return (error);

                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
                        return (SET_ERROR(ENXIO));
        }

        /*
         * Find the best uberblock.
         */
        vdev_uberblock_load(rvd, ub, &label);

        /*
         * If we weren’t able to find a single valid uberblock, return failure.
         */
        if (ub->ub_txg == 0) {
                nvlist_free(label);
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
        }

        /*
         * If the pool has an unsupported version we can’t open it.
         */
        if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
                nvlist_free(label);
                return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
        }

        if (ub->ub_version >= SPA_VERSION_FEATURES) {
                nvlist_t *features;

                /*
                 * If we weren’t able to find what’s necessary for reading the
                 * MOS in the label, return failure.
                 */
                if (label == NULL || nvlist_lookup_nvlist(label,
                    ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
                        nvlist_free(label);
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
                            ENXIO));
                }

                /*
                 * Update our in-core representation with the definitive values
                 * from the label.
                 */
                nvlist_free(spa->spa_label_features);
                VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
        }

        nvlist_free(label);

        /*
         * Look through entries in the label nvlist’s features_for_read. If
         * there is a feature listed there which we don’t understand then we
         * cannot open a pool.
         */
        if (ub->ub_version >= SPA_VERSION_FEATURES) {
                nvlist_t *unsup_feat;

                VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
                    0);
```

```

                for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
                    NULL); nvp != NULL;
                    nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
                        if (!zfeature_is_supported(nvpair_name(nvp))) {
                                VERIFY(nvlist_add_string(unsup_feat,
                                    nvpair_name(nvp), "") == 0);
                        }
                }

                if (!nvlist_empty(unsup_feat)) {
                        VERIFY(nvlist_add_nvlist(spa->spa_load_info,
                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
                        nvlist_free(unsup_feat);
                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
                            ENOTSUP));
                }

                nvlist_free(unsup_feat);
        }

        /*
         * If the vdev guid sum doesn’t match the uberblock, we have an
         * incomplete configuration.  We first check to see if the pool
         * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
         * If it is, defer the vdev_guid_sum check till later so we
         * can handle missing vdevs.
         */
        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
            &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
            rvd->vdev_guid_sum != ub->ub_guid_sum)
                return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));

        if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
                spa_try_repair(spa, config);
                spa_config_exit(spa, SCL_ALL, FTAG);
                nvlist_free(spa->spa_config_splitting);
                spa->spa_config_splitting = NULL;
        }

        /*
         * Initialize internal SPA structures.
         */
        spa->spa_state = POOL_STATE_ACTIVE;
        spa->spa_ubsync = spa->spa_uberblock;
        spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
            TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
        spa->spa_first_txg = spa->spa_last_ubsync_txg ?
            spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
        spa->spa_claim_max_txg = spa->spa_first_txg;
        spa->spa_prev_software_version = ub->ub_software_version;

        error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
        if (error)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;

        if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        if (spa_version(spa) >= SPA_VERSION_FEATURES) {
                boolean_t missing_feat_read = B_FALSE;
                nvlist_t *unsup_feat, *enabled_feat;

                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
                    &spa->spa_feat_for_read_obj) != 0) {
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
                }

                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
                    &spa->spa_feat_for_write_obj) != 0) {
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
                }

                if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
                    &spa->spa_feat_desc_obj) != 0) {
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
                }

                enabled_feat = fnvlist_alloc();
                unsup_feat = fnvlist_alloc();

                if (!spa_features_check(spa, B_FALSE,
                    unsup_feat, enabled_feat))
                        missing_feat_read = B_TRUE;

                if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
                        if (!spa_features_check(spa, B_TRUE,
                            unsup_feat, enabled_feat)) {
                                missing_feat_write = B_TRUE;
                        }
                }

                fnvlist_add_nvlist(spa->spa_load_info,
                    ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);

                if (!nvlist_empty(unsup_feat)) {
                        fnvlist_add_nvlist(spa->spa_load_info,
                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
                }

                fnvlist_free(enabled_feat);
                fnvlist_free(unsup_feat);

                if (!missing_feat_read) {
                        fnvlist_add_boolean(spa->spa_load_info,
                            ZPOOL_CONFIG_CAN_RDONLY);
                }

                /*
                 * If the state is SPA_LOAD_TRYIMPORT, our objective is
                 * twofold: to determine whether the pool is available for
                 * import in read-write mode and (if it is not) whether the
                 * pool is available for import in read-only mode. If the pool
                 * is available for import in read-write mode, it is displayed
                 * as available in userland; if it is not available for import
                 * in read-only mode, it is displayed as unavailable in
                 * userland. If the pool is available for import in read-only
                 * mode but not read-write mode, it is displayed as unavailable
                 * in userland with a special note that the pool is actually
                 * available for open in read-only mode.
                 *
                 * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
                 * missing a feature for write, we must first determine whether
                 * the pool can be opened read-only before returning to
                 * userland in order to know whether to display the
                 * abovementioned note.
                 */
                if (missing_feat_read || (missing_feat_write &&
                    spa_writeable(spa))) {
                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
```

```
                            ENOTSUP));
                }

                /*
                 * Load refcounts for ZFS features from disk into an in-memory
                 * cache during SPA initialization.
                 */
                for (spa_feature_t i = 0; i < SPA_FEATURES; i++) {
                        uint64_t refcount;

                        error = feature_get_refcount_from_disk(spa,
                            &spa_feature_table[i], &refcount);
                        if (error == 0) {
                                spa->spa_feat_refcount_cache[i] = refcount;
                        } else if (error == ENOTSUP) {
                                spa->spa_feat_refcount_cache[i] =
                                    SPA_FEATURE_DISABLED;
                        } else {
                                return (spa_vdev_err(rvd,
                                    VDEV_AUX_CORRUPT_DATA, EIO));
                        }
                }
        }

        if (spa_feature_is_active(spa, SPA_FEATURE_ENABLED_TXG)) {
                if (spa_dir_prop(spa, DMU_POOL_FEATURE_ENABLED_TXG,
                    &spa->spa_feat_enabled_txg_obj) != 0)
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        }

        spa->spa_is_initializing = B_TRUE;
        error = dsl_pool_open(spa->spa_dsl_pool);
        spa->spa_is_initializing = B_FALSE;
        if (error != 0)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        if (!mosconfig) {
                uint64_t hostid;
                nvlist_t *policy = NULL, *nvconfig;

                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

                if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
                    ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
                        char *hostname;
                        unsigned long myhostid = 0;

                        VERIFY(nvlist_lookup_string(nvconfig,
                            ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);

#ifdef  _KERNEL
                        myhostid = zone_get_hostid(NULL);
#else   /* _KERNEL */
                        /*
                         * We’re emulating the system’s hostid in userland, so
                         * we can’t use zone_get_hostid().
                         */
                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
#endif  /* _KERNEL */
                        if (hostid != 0 && myhostid != 0 &&
                            hostid != myhostid) {
                                nvlist_free(nvconfig);
                                cmn_err(CE_WARN, "pool ’%s’ could not be "
                                    "loaded as it was last accessed by "
                                    "another system (host: %s hostid: 0x%lx). "
                                    "See: http://illumos.org/msg/ZFS-8000-EY",
                                    spa_name(spa), hostname,
                                    (unsigned long)hostid);
                                return (SET_ERROR(EBADF));
                        }
                }
                if (nvlist_lookup_nvlist(spa->spa_config,
                    ZPOOL_REWIND_POLICY, &policy) == 0)
                        VERIFY(nvlist_add_nvlist(nvconfig,
                            ZPOOL_REWIND_POLICY, policy) == 0);

                spa_config_set(spa, nvconfig);
                spa_unload(spa);
                spa_deactivate(spa);
                spa_activate(spa, orig_mode);

                return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
        }

        if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
        if (error != 0)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        /*
         * Load the bit that tells us to use the new accounting function
         * (raid-z deflation).  If we have an older pool, this will not
         * be present.
         */
        error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
            &spa->spa_creation_version);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        /*
         * Load the persistent error log.  If we have an older pool, this will
         * not be present.
         */
        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
            &spa->spa_errlog_scrub);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        /*
         * Load the history object.  If we have an older pool, this
         * will not be present.
         */
        error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        /*
         * If we're assembling the pool from the split-off vdevs of
         * an existing pool, we don't want to attach the spares & cache
         * devices.
         */
```

```
        /*
         * Load any hot spares for this pool.
         */
        error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
                ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
                if (load_nvlist(spa, spa->spa_spares.sav_object,
                    &spa->spa_spares.sav_config) != 0)
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
                spa_load_spares(spa);
                spa_config_exit(spa, SCL_ALL, FTAG);
        } else if (error == 0) {
                spa->spa_spares.sav_sync = B_TRUE;
        }

        /*
         * Load any level 2 ARC devices for this pool.
         */
        error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
            &spa->spa_l2cache.sav_object);
        if (error != 0 && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
                ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
                if (load_nvlist(spa, spa->spa_l2cache.sav_object,
                    &spa->spa_l2cache.sav_config) != 0)
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
                spa_load_l2cache(spa);
                spa_config_exit(spa, SCL_ALL, FTAG);
        } else if (error == 0) {
                spa->spa_l2cache.sav_sync = B_TRUE;
        }

        spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);

        error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
        if (error && error != ENOENT)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        if (error == 0) {
                uint64_t autoreplace;

                spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
                spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
                spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
                spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
                spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
                spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
                    &spa->spa_dedup_ditto);

                spa->spa_autoreplace = (autoreplace != 0);
        }

        /*
         * If the 'autoreplace' property is set, then post a resource notifying
         * the ZFS DE that it should not issue any faults for unopenable
         * devices.  We also iterate over the vdevs, and post a sysevent for any
         * unopenable vdevs so that the normal autoreplace handler can take
         * over.
         */
        if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
                spa_check_removed(spa->spa_root_vdev);
                /*
                 * For the import case, this is done in spa_import(), because
                 * at this point we're using the spare definitions from
                 * the MOS config, not necessarily from the userland config.
                 */
                if (state != SPA_LOAD_IMPORT) {
                        spa_aux_check_removed(&spa->spa_spares);
                        spa_aux_check_removed(&spa->spa_l2cache);
                }
        }

        /*
         * Load the vdev state for all toplevel vdevs.
         */
        vdev_load(rvd);

        /*
         * Propagate the leaf DTLs we just loaded all the way up the tree.
         */
        spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
        vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
        spa_config_exit(spa, SCL_ALL, FTAG);

        /*
         * Load the DDTs (dedup tables).
         */
        error = ddt_load(spa);
        if (error != 0)
                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

        spa_update_dspace(spa);

        /*
         * Validate the config, using the MOS config to fill in any
         * information which might be missing.  If we fail to validate
         * the config then declare the pool unfit for use. If we're
         * assembling a pool from a split, the log is not transferred
         * over.
         */
        if (type != SPA_IMPORT_ASSEMBLE) {
                nvlist_t *nvconfig;

                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

                if (!spa_config_valid(spa, nvconfig)) {
                        nvlist_free(nvconfig);
                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
                            ENXIO));
                }
                nvlist_free(nvconfig);

                /*
                 * Now that we've validated the config, check the state of the
                 * root vdev.  If it can't be opened, it indicates one or
                 * more toplevel vdevs are faulted.
                 */
                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
                        return (SET_ERROR(ENXIO));

                if (spa_check_logs(spa)) {
                        *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
                }
```

```
        }

        if (missing_feat_write) {
                ASSERT(state == SPA_LOAD_TRYIMPORT);

                /*
                 * At this point, we know that we can open the pool in
                 * read-only mode but not read-write mode. We now have enough
                 * information and can return to userland.
                 */
                return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
        }

        /*
         * We've successfully opened the pool, verify that we're ready
         * to start pushing transactions.
         */
        if (state != SPA_LOAD_TRYIMPORT) {
                if (error = spa_load_verify(spa))
                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
                            error));
        }

        if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
            spa->spa_load_max_txg == UINT64_MAX)) {
                dmu_tx_t *tx;
                int need_update = B_FALSE;
                dsl_pool_t *dp = spa_get_dsl(spa);
#endif /* ! codereview */

                ASSERT(state != SPA_LOAD_TRYIMPORT);

                /*
                 * Claim log blocks that haven't been committed yet.
                 * This must all happen in a single txg.
                 * Note: spa_claim_max_txg is updated by spa_claim_notify(),
                 * invoked from zil_claim_log_block()'s i/o done callback.
                 * Price of rollback is that we abandon the log.
                 */
                spa->spa_claiming = B_TRUE;

                tx = dmu_tx_create_assigned(dp, spa_first_txg(spa));
                (void) dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
                tx = dmu_tx_create_assigned(spa_get_dsl(spa),
                    spa_first_txg(spa));
                (void) dmu_objset_find(spa_name(spa),
                    zil_claim, tx, DS_FIND_CHILDREN);
                dmu_tx_commit(tx);

                spa->spa_claiming = B_FALSE;

                spa_set_log_state(spa, SPA_LOG_GOOD);
                spa->spa_sync_on = B_TRUE;
                txg_sync_start(spa->spa_dsl_pool);

                /*
                 * Wait for all claims to sync.  We sync up to the highest
                 * claimed log block birth time so that claimed log blocks
                 * don't appear to be from the future.  spa_claim_max_txg
                 * will have been set for us by either zil_check_log_chain()
                 * (invoked from spa_check_logs()) or zil_claim() above.
                 */
                txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);

                /*
                 * If the config cache is stale, or we have uninitialized
                 * metaslabs (see spa_vdev_add()), then update the config.
                 *
                 * If this is a verbatim import, trust the current
                 * in-core spa_config and update the disk labels.
                 */
                if (config_cache_txg != spa->spa_config_txg ||
                    state == SPA_LOAD_IMPORT ||
                    state == SPA_LOAD_RECOVER ||
                    (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
                        need_update = B_TRUE;

                for (int c = 0; c < rvd->vdev_children; c++)
                        if (rvd->vdev_child[c]->vdev_ms_array == 0)
                                need_update = B_TRUE;

                /*
                 * Update the config cache asychronously in case we’re the
                 * root pool, in which case the config cache isn’t writable yet.
                 */
                if (need_update)
                        spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

                /*
                 * Check all DTLs to see if anything needs resilvering.
                 */
                if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
                    vdev_resilver_needed(rvd, NULL, NULL))
                        spa_async_request(spa, SPA_ASYNC_RESILVER);

                /*
                 * Log the fact that we booted up (so that we can detect if
                 * we rebooted in the middle of an operation).
                 */
                spa_history_log_version(spa, "open");

                /*
                 * Delete any inconsistent datasets.
                 */
                (void) dmu_objset_find(spa_name(spa),
                    dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

                /*
                 * Clean up any stale temporary dataset userrefs.
                 */
                dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
        }

        return (0);
}
```

______***unchanged_portion_omitted_***

```
**********************************************************
   33543 Wed May  6 08:47:28 2015
new/usr/src/uts/common/fs/zfs/sys/dmu.h
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
void byteswap_uint64_array(void *buf, size_t size);
void byteswap_uint32_array(void *buf, size_t size);
void byteswap_uint16_array(void *buf, size_t size);
void byteswap_uint8_array(void *buf, size_t size);
void zap_byteswap(void *buf, size_t size);
void zfs_oldacl_byteswap(void *buf, size_t size);
void zfs_acl_byteswap(void *buf, size_t size);
void zfs_znode_byteswap(void *buf, size_t size);

#define DS_FIND_SNAPSHOTS       (1<<0)
#define DS_FIND_CHILDREN        (1<<1)
#define DS_FIND_SERIALIZE       (1<<2)
#endif /* ! codereview */

/*
 * The maximum number of bytes that can be accessed as part of one
 * operation, including metadata.
 */
#define DMU_MAX_ACCESS (32 * 1024 * 1024) /* 32MB */
#define DMU_MAX_DELETEBLKCNT (20480) /* ~5MB of indirect blocks */

#define DMU_USERUSED_OBJECT     (-1ULL)
#define DMU_GROUPUSED_OBJECT    (-2ULL)

/*
 * artificial blkids for bonus buffer and spill blocks
 */
#define DMU_BONUS_BLKID         (-1ULL)
#define DMU_SPILL_BLKID         (-2ULL)
/*
 * Public routines to create, destroy, open, and close objsets.
 */
int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
int dmu_objset_own(const char *name, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp);
void dmu_objset_rele(objset_t *os, void *tag);
void dmu_objset_disown(objset_t *os, void *tag);
int dmu_objset_open_ds(struct dsl_dataset *ds, objset_t **osp);

void dmu_objset_evict_dbufs(objset_t *os);
int dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
    void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg);
int dmu_objset_clone(const char *name, const char *origin);
int dsl_destroy_snapshots_nvl(struct nvlist *snaps, boolean_t defer,
    struct nvlist *errlist);
int dmu_objset_snapshot_one(const char *fsname, const char *snapname);
int dmu_objset_snapshot_tmp(const char *, const char *, int);
int dmu_objset_find(char *name, int func(const char *, void *), void *arg,
    int flags);
```

```
void dmu_objset_byteswap(void *buf, size_t size);
int dsl_dataset_rename_snapshot(const char *fsname,
    const char *oldsnapname, const char *newsnapname, boolean_t recursive);

typedef struct dmu_buf {
        uint64_t db_object;             /* object that this buffer is part of */
        uint64_t db_offset;             /* byte offset in this object */
        uint64_t db_size;               /* size of buffer in bytes */
        void *db_data;                  /* data in buffer */
} dmu_buf_t;

/*
 * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
 */
#define DMU_POOL_DIRECTORY_OBJECT       1
#define DMU_POOL_CONFIG                 "config"
#define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
#define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
#define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
#define DMU_POOL_FEATURE_ENABLED_TXG    "feature_enabled_txg"
#define DMU_POOL_ROOT_DATASET           "root_dataset"
#define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
#define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
#define DMU_POOL_ERRLOG_LAST            "errlog_last"
#define DMU_POOL_SPARES                 "spares"
#define DMU_POOL_DEFLATE                "deflate"
#define DMU_POOL_HISTORY                "history"
#define DMU_POOL_PROPS                  "pool_props"
#define DMU_POOL_L2CACHE                "l2cache"
#define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
#define DMU_POOL_DDT                    "DDT-%s-%s-%s"
#define DMU_POOL_DDT_STATS              "DDT-statistics"
#define DMU_POOL_CREATION_VERSION       "creation_version"
#define DMU_POOL_SCAN                   "scan"
#define DMU_POOL_FREE_BPOBJ             "free_bpobj"
#define DMU_POOL_BPTREE_OBJ             "bptree_obj"
#define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"

/*
 * Allocate an object from this objset.  The range of object numbers
 * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
 *
 * The transaction must be assigned to a txg.  The newly allocated
 * object will be "held" in the transaction (ie. you can modify the
 * newly allocated object in this transaction).
 *
 * dmu_object_alloc() chooses an object and returns it in *objectp.
 *
 * dmu_object_claim() allocates a specific object number.  If that
 * number is already allocated, it fails and returns EEXIST.
 *
 * Return 0 on success, or ENOSPC or EEXIST as specified above.
 */
uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
    int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
    int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
    int blocksize, dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *txp);

/*
 * Free an object from this objset.
 *
 * The object’s data will be freed as well (ie. you don’t need to call
 * dmu_free(object, 0, -1, tx)).
 *
 * The object need not be held in the transaction.
 *
 * If there are any holds on this object's buffers (via dmu_buf_hold()),
 * or tx holds on the object (via dmu_tx_hold_object()), you can not
 * free it; it fails and returns EBUSY.
 *
 * If the object is not allocated, it fails and returns ENOENT.
 *
 * Return 0 on success, or EBUSY or ENOENT as specified above.
 */
int dmu_object_free(objset_t *os, uint64_t object, dmu_tx_t *tx);

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

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

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

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

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

/*
 * Decide how to write a block: checksum, compression, number of copies, etc.
 */
#define WP_NOFILL       0x1
#define WP_DMU_SYNC     0x2
#define WP_SPILL        0x4
```

```
void dmu_write_policy(objset_t *os, struct dnode *dn, int level, int wp,
    struct zio_prop *zp);
/*
 * The bonus data is accessed more or less like a regular buffer.
 * You must dmu_bonus_hold() to get the buffer, which will give you a
 * dmu_buf_t with db_offset==-1ULL, and db_size = the size of the bonus
 * data.  As with any normal buffer, you must call dmu_buf_read() to
 * read db_data, dmu_buf_will_dirty() before modifying it, and the
 * object must be held in an assigned transaction before calling
 * dmu_buf_will_dirty.  You may use dmu_buf_set_user() on the bonus
 * buffer as well.  You must release your hold with dmu_buf_rele().
 *
 * Returns ENOENT, EIO, or 0.
 */
int dmu_bonus_hold(objset_t *os, uint64_t object, void *tag, dmu_buf_t **);
int dmu_bonus_max(void);
int dmu_set_bonus(dmu_buf_t *, int, dmu_tx_t *);
int dmu_set_bonustype(dmu_buf_t *, dmu_object_type_t, dmu_tx_t *);
dmu_object_type_t dmu_get_bonustype(dmu_buf_t *);
int dmu_rm_spill(objset_t *, uint64_t, dmu_tx_t *);

/*
 * Special spill buffer support used by "SA" framework
 */

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

/*
 * Obtain the DMU buffer from the specified object which contains the
 * specified offset.  dmu_buf_hold() puts a "hold" on the buffer, so
 * that it will remain in memory.  You must release the hold with
 * dmu_buf_rele().  You musn't access the dmu_buf_t after releasing your
 * hold.  You must have a hold on any dmu_buf_t* you pass to the DMU.
 *
 * You must call dmu_buf_read, dmu_buf_will_dirty, or dmu_buf_will_fill
 * on the returned buffer before reading or writing the buffer's
 * db_data.  The comments for those routines describe what particular
 * operations are valid after calling them.
 *
 * The object number must be a valid, allocated object number.
 */
int dmu_buf_hold(objset_t *os, uint64_t object, uint64_t offset,
    void *tag, dmu_buf_t **, int flags);

/*
 * Add a reference to a dmu buffer that has already been held via
 * dmu_buf_hold() in the current context.
 */
void dmu_buf_add_ref(dmu_buf_t *db, void* tag);

/*
 * Attempt to add a reference to a dmu buffer that is in an unknown state,
 * using a pointer that may have been invalidated by eviction processing.
 * The request will succeed if the passed in dbuf still represents the
 * same os/object/blkid, is ineligible for eviction, and has at least
 * one hold by a user other than the syncer.
 */
boolean_t dmu_buf_try_add_ref(dmu_buf_t *, objset_t *os, uint64_t object,
    uint64_t blkid, void *tag);

void dmu_buf_rele(dmu_buf_t *db, void *tag);
uint64_t dmu_buf_refcount(dmu_buf_t *db);
```

```
/*
 * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
 * range of an object.  A pointer to an array of dmu_buf_t*’s is
 * returned (in *dbpp).
 *
 * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*’s, and
 * frees the array.  The hold on the array of buffers MUST be released
 * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
 * individually with dmu_buf_rele.
 */
int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
    uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

typedef void dmu_buf_evict_func_t(void *user_ptr);

/*
 * A DMU buffer user object may be associated with a dbuf for the
 * duration of its lifetime.  This allows the user of a dbuf (client)
 * to attach private data to a dbuf (e.g. in-core only data such as a
 * dnode_children_t, zap_t, or zap_leaf_t) and be optionally notified
 * when that dbuf has been evicted.  Clients typically respond to the
 * eviction notification by freeing their private data, thus ensuring
 * the same lifetime for both dbuf and private data.
 *
 * The mapping from a dmu_buf_user_t to any client private data is the
 * client’s responsibility.  All current consumers of the API with private
 * data embed a dmu_buf_user_t as the first member of the structure for
 * their private data.  This allows conversions between the two types
 * with a simple cast.  Since the DMU buf user API never needs access
 * to the private data, other strategies can be employed if necessary
 * or convenient for the client (e.g. using container_of() to do the
 * conversion for private data that cannot have the dmu_buf_user_t as
 * its first member).
 *
 * Eviction callbacks are executed without the dbuf mutex held or any
 * other type of mechanism to guarantee that the dbuf is still available.
 * For this reason, users must assume the dbuf has already been freed
 * and not reference the dbuf from the callback context.
 *
 * Users requesting "immediate eviction" are notified as soon as the dbuf
 * is only referenced by dirty records (dirties == holds).  Otherwise the
 * notification occurs after eviction processing for the dbuf begins.
 */
typedef struct dmu_buf_user {
        /*
         * Asynchronous user eviction callback state.
         */
        taskq_ent_t     dbu_tqent;

        /* This instance’s eviction function pointer. */
        dmu_buf_evict_func_t *dbu_evict_func;
#ifdef ZFS_DEBUG
        /*
         * Pointer to user’s dbuf pointer.  NULL for clients that do
         * not associate a dbuf with their user data.
         *
         * The dbuf pointer is cleared upon eviction so as to catch
         * use-after-evict bugs in clients.
         */
        dmu_buf_t **dbu_clear_on_evict_dbufp;
#endif
} dmu_buf_user_t;

/*
 * Initialize the given dmu_buf_user_t instance with the eviction function
```

```
 * evict_func, to be called when the user is evicted.
 *
 * NOTE: This function should only be called once on a given dmu_buf_user_t.
 *       To allow enforcement of this, dbu must already be zeroed on entry.
 */
#ifdef __lint
/* Very ugly, but it beats issuing suppression directives in many Makefiles. */
extern void
dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
    dmu_buf_t **clear_on_evict_dbufp);
#else /* __lint */
inline void
dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
    dmu_buf_t **clear_on_evict_dbufp)
{
        ASSERT(dbu->dbu_evict_func == NULL);
        ASSERT(evict_func != NULL);
        dbu->dbu_evict_func = evict_func;
#ifdef ZFS_DEBUG
        dbu->dbu_clear_on_evict_dbufp = clear_on_evict_dbufp;
#endif
}
#endif /* __lint */

/*
 * Attach user data to a dbuf and mark it for normal (when the dbuf’s
 * data is cleared or its reference count goes to zero) eviction processing.
 *
 * Returns NULL on success, or the existing user if another user currently
 * owns the buffer.
 */
void *dmu_buf_set_user(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Attach user data to a dbuf and mark it for immediate (its dirty and
 * reference counts are equal) eviction processing.
 *
 * Returns NULL on success, or the existing user if another user currently
 * owns the buffer.
 */
void *dmu_buf_set_user_ie(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Replace the current user of a dbuf.
 *
 * If given the current user of a dbuf, replaces the dbuf’s user with
 * "new_user" and returns the user data pointer that was replaced.
 * Otherwise returns the current, and unmodified, dbuf user pointer.
 */
void *dmu_buf_replace_user(dmu_buf_t *db,
    dmu_buf_user_t *old_user, dmu_buf_user_t *new_user);

/*
 * Remove the specified user data for a DMU buffer.
 *
 * Returns the user that was removed on success, or the current user if
 * another user currently owns the buffer.
 */
void *dmu_buf_remove_user(dmu_buf_t *db, dmu_buf_user_t *user);

/*
 * Returns the user data (dmu_buf_user_t *) associated with this dbuf.
 */
void *dmu_buf_get_user(dmu_buf_t *db);

/* Block until any in-progress dmu buf user evictions complete. */
```

```
void dmu_buf_user_evict_wait(void);

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

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

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

/*
 * You must create a transaction, then hold the objects which you will
 * (or might) modify as part of this transaction.  Then you must assign
 * the transaction to a transaction group.  Once the transaction has
 * been assigned, you can modify buffers which belong to held objects as
 * part of this transaction.  You can’t modify buffers before the
 * transaction has been assigned; you can’t modify buffers which don’t
 * belong to objects which this transaction holds; you can’t hold
 * objects once the transaction has been assigned.  You may hold an
 * object which you are going to free (with dmu_object_free()), but you
 * don’t have to.
 *
 * You can abort the transaction before it has been assigned.
 *
 * Note that you may hold buffers (with dmu_buf_hold) at any time,
 * regardless of transaction state.
 */

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

dmu_tx_t *dmu_tx_create(objset_t *os);
void dmu_tx_hold_write(dmu_tx_t *tx, uint64_t object, uint64_t off, int len);
void dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off,
    uint64_t len);
void dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name);
void dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object);
void dmu_tx_hold_sa(dmu_tx_t *tx, struct sa_handle *hdl, boolean_t may_grow);
void dmu_tx_hold_sa_create(dmu_tx_t *tx, int total_size);
void dmu_tx_abort(dmu_tx_t *tx);
int dmu_tx_assign(dmu_tx_t *tx, enum txg_how txg_how);
void dmu_tx_wait(dmu_tx_t *tx);
void dmu_tx_commit(dmu_tx_t *tx);
void dmu_tx_mark_netfree(dmu_tx_t *tx);

/*
 * To register a commit callback, dmu_tx_callback_register() must be called.
 *
 * dcb_data is a pointer to caller private data that is passed on as a
 * callback parameter. The caller is responsible for properly allocating and
 * freeing it.
 *
 * When registering a callback, the transaction must be already created, but
 * it cannot be committed or aborted. It can be assigned to a txg or not.
```

```
 *
 * The callback will be called after the transaction has been safely written
 * to stable storage and will also be called if the dmu_tx is aborted.
 * If there is any error which prevents the transaction from being committed to
 * disk, the callback will be called with a value of error != 0.
 */
typedef void dmu_tx_callback_func_t(void *dcb_data, int error);

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

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

/*
 * Convenience functions.
 *
 * Canfail routines will return 0 on success, or an errno if there is a
 * nonrecoverable I/O error.
 */
#define DMU_READ_PREFETCH       0 /* prefetch */
#define DMU_READ_NO_PREFETCH    1 /* don’t prefetch */
int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        void *buf, uint32_t flags);
void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        const void *buf, dmu_tx_t *tx);
void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
        dmu_tx_t *tx);
int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
int dmu_read_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size);
int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
    dmu_tx_t *tx);
int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
    dmu_tx_t *tx);
int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
    uint64_t size, struct page *pp, dmu_tx_t *tx);
struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
void dmu_return_arcbuf(struct arc_buf *buf);
void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
    dmu_tx_t *tx);
int dmu_xuio_init(struct xuio *uio, int niov);
void dmu_xuio_fini(struct xuio *uio);
int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
    size_t n);
int dmu_xuio_cnt(struct xuio *uio);
struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
void dmu_xuio_clear(struct xuio *uio, int i);
void xuio_stat_wbuf_copied();
void xuio_stat_wbuf_nocopy();

extern int zfs_prefetch_disable;
extern int zfs_max_recordsize;

/*
 * Asynchronously try to read in the data.
 */
void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
    uint64_t len);
```

```
typedef struct dmu_object_info {
        /* All sizes are in bytes unless otherwise indicated. */
        uint32_t doi_data_block_size;
        uint32_t doi_metadata_block_size;
        dmu_object_type_t doi_type;
        dmu_object_type_t doi_bonus_type;
        uint64_t doi_bonus_size;
        uint8_t doi_indirection;                /* 2 = dnode->indirect->data */
        uint8_t doi_checksum;
        uint8_t doi_compress;
        uint8_t doi_nblkptr;
        uint8_t doi_pad[4];
        uint64_t doi_physical_blocks_512;       /* data + metadata, 512b blks */
        uint64_t doi_max_offset;
        uint64_t doi_fill_count;                /* number of non-empty blocks */
} dmu_object_info_t;

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

typedef struct dmu_object_type_info {
        dmu_object_byteswap_t   ot_byteswap;
        boolean_t               ot_metadata;
        char                    *ot_name;
} dmu_object_type_info_t;

typedef struct dmu_object_byteswap_info {
        arc_byteswap_func_t     *ob_func;
        char                    *ob_name;
} dmu_object_byteswap_info_t;

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

/*
 * 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);

/*
 * Check if a DMU object has any dirty blocks. If so, sync out
 * all pending transaction groups. Otherwise, this function
 * does not alter DMU state. This could be improved to only sync
 * out the necessary transaction groups for this particular
 * object.
 */
int dmu_object_wait_synced(objset_t *os, uint64_t object);

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

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

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

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

extern int zfs_mdcomp_disable;

#ifdef  __cplusplus
}
#endif

#endif  /* _SYS_DMU_H */
```

****
6101 Wed May 6 08:47:28 2015
new/usr/src/uts/common/fs/zfs/sys/dmu_objset.h
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****

______unchanged_portion_omitted_

```
#define DMU_META_OBJSET         0
#define DMU_META_DNODE_OBJECT   0
#define DMU_OBJECT_IS_SPECIAL(obj) ((int64_t)(obj) <= 0)
#define DMU_META_DNODE(os)      ((os)->os_meta_dnode.dnh_dnode)
#define DMU_USERUSED_DNODE(os)  ((os)->os_userused_dnode.dnh_dnode)
#define DMU_GROUPUSED_DNODE(os) ((os)->os_groupused_dnode.dnh_dnode)

#define DMU_OS_IS_L2CACHEABLE(os)                               \
        ((os)->os_secondary_cache == ZFS_CACHE_ALL ||           \
        (os)->os_secondary_cache == ZFS_CACHE_METADATA)

#define DMU_OS_IS_L2COMPRESSIBLE(os)    (zfs_mdcomp_disable == B_FALSE)

/* called from zpl */
int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
int dmu_objset_own(const char *name, dmu_objset_type_t type,
    boolean_t readonly, void *tag, objset_t **osp);
int dmu_objset_own_obj(struct dsl_pool *dp, uint64_t obj,
    dmu_objset_type_t type, boolean_t readonly, void *tag, objset_t **osp);
#endif /* ! codereview */
void dmu_objset_refresh_ownership(objset_t *os, void *tag);
void dmu_objset_rele(objset_t *os, void *tag);
void dmu_objset_disown(objset_t *os, void *tag);
int dmu_objset_from_ds(struct dsl_dataset *ds, objset_t **osp);

void dmu_objset_stats(objset_t *os, nvlist_t *nv);
void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);
void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
    uint64_t *usedobjsp, uint64_t *availobjsp);
uint64_t dmu_objset_fsid_guid(objset_t *os);
int dmu_objset_find_dp(struct dsl_pool *dp, uint64_t ddobj,
    int func(struct dsl_pool *, struct dsl_dataset *, void *),
    void *arg, int flags);
int dmu_objset_prefetch(const char *name, void *arg);
void dmu_objset_evict_dbufs(objset_t *os);
timestruc_t dmu_objset_snap_cmtime(objset_t *os);

/* called from dsl */
void dmu_objset_sync(objset_t *os, zio_t *zio, dmu_tx_t *tx);
boolean_t dmu_objset_is_dirty(objset_t *os, uint64_t txg);
objset_t *dmu_objset_create_impl(spa_t *spa, struct dsl_dataset *ds,
    blkptr_t *bp, dmu_objset_type_t type, dmu_tx_t *tx);
int dmu_objset_open_impl(spa_t *spa, struct dsl_dataset *ds, blkptr_t *bp,
    objset_t **osp);
void dmu_objset_evict(objset_t *os);
void dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx);
void dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx);
boolean_t dmu_objset_userused_enabled(objset_t *os);
int dmu_objset_userspace_upgrade(objset_t *os);
```

```
boolean_t dmu_objset_userspace_present(objset_t *os);
int dmu_fsname(const char *snapname, char *buf);

void dmu_objset_evict_done(objset_t *os);

void dmu_objset_init(void);
void dmu_objset_fini(void);

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_DMU_OBJSET_H */
```

```
**********************************************************
    5482 Wed May  6 08:47:28 2015
new/usr/src/uts/common/fs/zfs/sys/dsl_pool.h
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
int dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp);
int dsl_pool_open(dsl_pool_t *dp);
void dsl_pool_close(dsl_pool_t *dp);
dsl_pool_t *dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg);
void dsl_pool_sync(dsl_pool_t *dp, uint64_t txg);
void dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg);
int dsl_pool_sync_context(dsl_pool_t *dp);
uint64_t dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree);
uint64_t dsl_pool_adjustedfree(dsl_pool_t *dp, boolean_t netfree);
void dsl_pool_dirty_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
void dsl_pool_undirty_space(dsl_pool_t *dp, int64_t space, uint64_t txg);
void dsl_free(dsl_pool_t *dp, uint64_t txg, const blkptr_t *bpp);
void dsl_free_sync(zio_t *pio, dsl_pool_t *dp, uint64_t txg,
    const blkptr_t *bpp);
void dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx);
void dsl_pool_mos_diduse_space(dsl_pool_t *dp,
    int64_t used, int64_t comp, int64_t uncomp);
void dsl_pool_config_enter(dsl_pool_t *dp, void *tag);
void dsl_pool_config_exit(dsl_pool_t *dp, void *tag);
boolean_t dsl_pool_config_held(dsl_pool_t *dp);
boolean_t dsl_pool_config_held_writer(dsl_pool_t *dp);
#endif /* ! codereview */
boolean_t dsl_pool_need_dirty_delay(dsl_pool_t *dp);

taskq_t *dsl_pool_vnrele_taskq(dsl_pool_t *dp);

int dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, uint64_t now, dmu_tx_t *tx);
int dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj,
    const char *tag, dmu_tx_t *tx);
void dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp);
int dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **);
int dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp);
void dsl_pool_rele(dsl_pool_t *dp, void *tag);

#ifdef  __cplusplus
}
#endif

#endif /* _SYS_DSL_POOL_H */
```

****
5825 Wed May 6 08:47:28 2015
new/usr/src/uts/common/fs/zfs/sys/vdev.h
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****
______*unchanged_portion_omitted_*

```
extern boolean_t zfs_nocacheflush;

extern int vdev_open(vdev_t *);
extern void vdev_open_children(vdev_t *);
extern boolean_t vdev_uses_zvols(vdev_t *);
extern int vdev_validate(vdev_t *, boolean_t);
extern void vdev_close(vdev_t *);
extern int vdev_create(vdev_t *, uint64_t txg, boolean_t isreplace);
extern void vdev_reopen(vdev_t *);
extern int vdev_validate_aux(vdev_t *vd);
extern zio_t *vdev_probe(vdev_t *vd, zio_t *pio);

extern boolean_t vdev_is_bootable(vdev_t *vd);
extern vdev_t *vdev_lookup_top(spa_t *spa, uint64_t vdev);
extern vdev_t *vdev_lookup_by_guid(vdev_t *vd, uint64_t guid);
extern int vdev_count_leaves(spa_t *spa);
#endif /* ! codereview */
extern void vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t d,
    uint64_t txg, uint64_t size);
extern boolean_t vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t d,
    uint64_t txg, uint64_t size);
extern boolean_t vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t d);
extern void vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg,
    int scrub_done);
extern boolean_t vdev_dtl_required(vdev_t *vd);
extern boolean_t vdev_resilver_needed(vdev_t *vd,
    uint64_t *minp, uint64_t *maxp);

extern void vdev_hold(vdev_t *);
extern void vdev_rele(vdev_t *);

extern int vdev_metaslab_init(vdev_t *vd, uint64_t txg);
extern void vdev_metaslab_fini(vdev_t *vd);
extern void vdev_metaslab_set_size(vdev_t *);
extern void vdev_expand(vdev_t *vd, uint64_t txg);
extern void vdev_split(vdev_t *vd);
extern void vdev_deadman(vdev_t *vd);


extern void vdev_get_stats(vdev_t *vd, vdev_stat_t *vs);
extern void vdev_clear_stats(vdev_t *vd);
extern void vdev_stat_update(zio_t *zio, uint64_t psize);
extern void vdev_scan_stat_init(vdev_t *vd);
extern void vdev_propagate_state(vdev_t *vd);
extern void vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state,
    vdev_aux_t aux);

extern void vdev_space_update(vdev_t *vd,
    int64_t alloc_delta, int64_t defer_delta, int64_t space_delta);
```

```
extern uint64_t vdev_psize_to_asize(vdev_t *vd, uint64_t psize);

extern int vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux);
extern int vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux);
extern int vdev_online(spa_t *spa, uint64_t guid, uint64_t flags,
    vdev_state_t *);
extern int vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags);
extern void vdev_clear(spa_t *spa, vdev_t *vd);

extern boolean_t vdev_is_dead(vdev_t *vd);
extern boolean_t vdev_readable(vdev_t *vd);
extern boolean_t vdev_writeable(vdev_t *vd);
extern boolean_t vdev_allocatable(vdev_t *vd);
extern boolean_t vdev_accessible(vdev_t *vd, zio_t *zio);

extern void vdev_cache_init(vdev_t *vd);
extern void vdev_cache_fini(vdev_t *vd);
extern boolean_t vdev_cache_read(zio_t *zio);
extern void vdev_cache_write(zio_t *zio);
extern void vdev_cache_purge(vdev_t *vd);

extern void vdev_queue_init(vdev_t *vd);
extern void vdev_queue_fini(vdev_t *vd);
extern zio_t *vdev_queue_io(zio_t *zio);
extern void vdev_queue_io_done(zio_t *zio);

extern void vdev_config_dirty(vdev_t *vd);
extern void vdev_config_clean(vdev_t *vd);
extern int vdev_config_sync(vdev_t **svd, int svdcount, uint64_t txg,
    boolean_t);

extern void vdev_state_dirty(vdev_t *vd);
extern void vdev_state_clean(vdev_t *vd);

typedef enum vdev_config_flag {
        VDEV_CONFIG_SPARE = 1 << 0,
        VDEV_CONFIG_L2CACHE = 1 << 1,
        VDEV_CONFIG_REMOVING = 1 << 2
} vdev_config_flag_t;

extern void vdev_top_config_generate(spa_t *spa, nvlist_t *config);
extern nvlist_t *vdev_config_generate(spa_t *spa, vdev_t *vd,
    boolean_t getstats, vdev_config_flag_t flags);

/*
 * Label routines
 */
struct uberblock;
extern uint64_t vdev_label_offset(uint64_t psize, int l, uint64_t offset);
extern int vdev_label_number(uint64_t psise, uint64_t offset);
extern nvlist_t *vdev_label_read_config(vdev_t *vd, uint64_t txg);
extern void vdev_uberblock_load(vdev_t *, struct uberblock *, nvlist_t **);

typedef enum {
        VDEV_LABEL_CREATE,      /* create/add a new device */
        VDEV_LABEL_REPLACE,     /* replace an existing device */
        VDEV_LABEL_SPARE,       /* add a new hot spare */
        VDEV_LABEL_REMOVE,      /* remove an existing device */
        VDEV_LABEL_L2CACHE,     /* add an L2ARC cache device */
        VDEV_LABEL_SPLIT        /* generating new label for split-off dev */
} vdev_labeltype_t;

extern int vdev_label_init(vdev_t *vd, uint64_t txg, vdev_labeltype_t reason);

#ifdef  __cplusplus
```

```
}
#endif

#endif  /* _SYS_VDEV_H */
```

```
**********************************************************
   15328 Wed May  6 08:47:28 2015
new/usr/src/uts/common/fs/zfs/sys/zil.h
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
/*
 * 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 by Delphix. All rights reserved.
 */

/* Portions Copyright 2010 Robert Milkowski */

#ifndef _SYS_ZIL_H
#define _SYS_ZIL_H

#include <sys/types.h>
#include <sys/spa.h>
#include <sys/zio.h>
#include <sys/dmu.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct dsl_pool;
struct dsl_dataset;

#endif /* ! codereview */
/*
 * Intent log format:
 *
 * Each objset has its own intent log.  The log header (zil_header_t)
 * for objset N's intent log is kept in the Nth object of the SPA's
 * intent_log objset.  The log header points to a chain of log blocks,
 * each of which contains log records (i.e., transactions) followed by
 * a log block trailer (zil_trailer_t).  The format of a log record
```

```
 * depends on the record (or transaction) type, but all records begin
 * with a common structure that defines the type, length, and txg.
 */

/*
 * Intent log header - this on disk structure holds fields to manage
 * the log.  All fields are 64 bit to easily handle cross architectures.
 */
typedef struct zil_header {
        uint64_t zh_claim_txg;  /* txg in which log blocks were claimed */
        uint64_t zh_replay_seq; /* highest replayed sequence number */
        blkptr_t zh_log;        /* log chain */
        uint64_t zh_claim_blk_seq; /* highest claimed block sequence number */
        uint64_t zh_flags;      /* header flags */
        uint64_t zh_claim_lr_seq; /* highest claimed lr sequence number */
        uint64_t zh_pad[3];
} zil_header_t;

/*
 * zh_flags bit settings
 */
#define ZIL_REPLAY_NEEDED       0x1     /* replay needed - internal only */
#define ZIL_CLAIM_LR_SEQ_VALID  0x2     /* zh_claim_lr_seq field is valid */

/*
 * Log block chaining.
 *
 * Log blocks are chained together. Originally they were chained at the
 * end of the block. For performance reasons the chain was moved to the
 * beginning of the block which allows writes for only the data being used.
 * The older position is supported for backwards compatability.
 *
 * The zio_eck_t contains a zec_cksum which for the intent log is
 * the sequence number of this log block. A seq of 0 is invalid.
 * The zec_cksum is checked by the SPA against the sequence
 * number passed in the blk_cksum field of the blkptr_t
 */
typedef struct zil_chain {
        uint64_t zc_pad;
        blkptr_t zc_next_blk;   /* next block in chain */
        uint64_t zc_nused;      /* bytes in log block used */
        zio_eck_t zc_eck;       /* block trailer */
} zil_chain_t;

#define ZIL_MIN_BLKSZ   4096ULL

/*
 * The words of a log block checksum.
 */
#define ZIL_ZC_GUID_0   0
#define ZIL_ZC_GUID_1   1
#define ZIL_ZC_OBJSET   2
#define ZIL_ZC_SEQ      3

typedef enum zil_create {
        Z_FILE,
        Z_DIR,
        Z_XATTRDIR,
} zil_create_t;

/*
 * size of xvattr log section.
 * its composed of lr_attr_t + xvattr bitmap + 2 64 bit timestamps
 * for create time and a single 64 bit integer for all of the attributes,
 * and 4 64 bit integers (32 bytes) for the scanstamp.
 *
 */

#define ZIL_XVAT_SIZE(mapsize) \
        sizeof (lr_attr_t) + (sizeof (uint32_t) * (mapsize - 1)) + \
        (sizeof (uint64_t) * 7)

/*
 * Size of ACL in log.  The ACE data is padded out to properly align
 * on 8 byte boundary.
 */

#define ZIL_ACE_LENGTH(x)       (roundup(x, sizeof (uint64_t)))

/*
 * Intent log transaction types and record structures
 */
#define TX_CREATE               1       /* Create file */
#define TX_MKDIR                2       /* Make directory */
#define TX_MKXATTR              3       /* Make XATTR directory */
#define TX_SYMLINK              4       /* Create symbolic link to a file */
#define TX_REMOVE               5       /* Remove file */
#define TX_RMDIR                6       /* Remove directory */
#define TX_LINK                 7       /* Create hard link to a file */
#define TX_RENAME               8       /* Rename a file */
#define TX_WRITE                9       /* File write */
#define TX_TRUNCATE             10      /* Truncate a file */
#define TX_SETATTR              11      /* Set file attributes */
#define TX_ACL_V0               12      /* Set old formatted ACL */
#define TX_ACL                  13      /* Set ACL */
#define TX_CREATE_ACL           14      /* create with ACL */
#define TX_CREATE_ATTR          15      /* create + attrs */
#define TX_CREATE_ACL_ATTR      16      /* create with ACL + attrs */
#define TX_MKDIR_ACL            17      /* mkdir with ACL */
#define TX_MKDIR_ATTR           18      /* mkdir with attr */
#define TX_MKDIR_ACL_ATTR       19      /* mkdir with ACL + attrs */
#define TX_WRITE2               20      /* dmu_sync EALREADY write */
#define TX_MAX_TYPE             21      /* Max transaction type */

/*
 * The transactions for mkdir, symlink, remove, rmdir, link, and rename
 * may have the following bit set, indicating the original request
 * specified case-insensitive handling of names.
 */
#define TX_CI   ((uint64_t)0x1 << 63) /* case-insensitive behavior requested */

/*
 * Transactions for write, truncate, setattr, acl_v0, and acl can be logged
 * out of order.  For convenience in the code, all such records must have
 * lr_foid at the same offset.
 */
#define TX_OOO(txtype)                  \
        ((txtype) == TX_WRITE ||        \
        (txtype) == TX_TRUNCATE ||      \
        (txtype) == TX_SETATTR ||       \
        (txtype) == TX_ACL_V0 ||        \
        (txtype) == TX_ACL ||           \
        (txtype) == TX_WRITE2)

/*
 * Format of log records.
 * The fields are carefully defined to allow them to be aligned
 * and sized the same on sparc & intel architectures.
 * Each log record has a common structure at the beginning.
 *
 * The log record on disk (lrc_seq) holds the sequence number of all log
 * records which is used to ensure we don’t replay the same record.
```

```
 */
typedef struct {                        /* common log record header */
        uint64_t        lrc_txtype;     /* intent log transaction type */
        uint64_t        lrc_reclen;     /* transaction record length */
        uint64_t        lrc_txg;        /* dmu transaction group number */
        uint64_t        lrc_seq;        /* see comment above */
} lr_t;

/*
 * Common start of all out-of-order record types (TX_OOO() above).
 */
typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_foid;        /* object id */
} lr_ooo_t;

/*
 * Handle option extended vattr attributes.
 *
 * Whenever new attributes are added the version number
 * will need to be updated as will code in
 * zfs_log.c and zfs_replay.c
 */
typedef struct {
        uint32_t        lr_attr_masksize; /* number of elements in array */
        uint32_t        lr_attr_bitmap; /* First entry of array */
        /* remainder of array and any additional fields */
} lr_attr_t;

/*
 * log record for creates without optional ACL.
 * This log record does support optional xvattr_t attributes.
 */
typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_doid;        /* object id of directory */
        uint64_t        lr_foid;        /* object id of created file object */
        uint64_t        lr_mode;        /* mode of object */
        uint64_t        lr_uid;         /* uid of object */
        uint64_t        lr_gid;         /* gid of object */
        uint64_t        lr_gen;         /* generation (txg of creation) */
        uint64_t        lr_crtime[2];   /* creation time */
        uint64_t        lr_rdev;        /* rdev of object to create */
        /* name of object to create follows this */
        /* for symlinks, link content follows name */
        /* for creates with xvattr data, the name follows the xvattr info */
} lr_create_t;

/*
 * FUID ACL record will be an array of ACEs from the original ACL.
 * If this array includes ephemeral IDs, the record will also include
 * an array of log-specific FUIDs to replace the ephemeral IDs.
 * Only one copy of each unique domain will be present, so the log-specific
 * FUIDs will use an index into a compressed domain table.  On replay this
 * information will be used to construct real FUIDs (and bypass idmap,
 * since it may not be available).
 */

/*
 * Log record for creates with optional ACL
 * This log record is also used for recording any FUID
 * information needed for replaying the create.  If the
 * file doesn’t have any actual ACEs then the lr_aclcnt
 * would be zero.
 *
 * After lr_acl_flags, there are a lr_acl_bytes number of variable sized ace’s.
 * If create is also setting xvattr’s, then acl data follows xvattr.
 * If ACE FUIDs are needed then they will follow the xvattr_t.  Following
 * the FUIDs will be the domain table information.  The FUIDs for the owner
 * and group will be in lr_create.  Name follows ACL data.
 */
typedef struct {
        lr_create_t     lr_create;      /* common create portion */
        uint64_t        lr_aclcnt;      /* number of ACEs in ACL */
        uint64_t        lr_domcnt;      /* number of unique domains */
        uint64_t        lr_fuidcnt;     /* number of real fuids */
        uint64_t        lr_acl_bytes;   /* number of bytes in ACL */
        uint64_t        lr_acl_flags;   /* ACL flags */
} lr_acl_create_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_doid;        /* obj id of directory */
        /* name of object to remove follows this */
} lr_remove_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_doid;        /* obj id of directory */
        uint64_t        lr_link_obj;    /* obj id of link */
        /* name of object to link follows this */
} lr_link_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_sdoid;       /* obj id of source directory */
        uint64_t        lr_tdoid;       /* obj id of target directory */
        /* 2 strings: names of source and destination follow this */
} lr_rename_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_foid;        /* file object to write */
        uint64_t        lr_offset;      /* offset to write to */
        uint64_t        lr_length;      /* user data length to write */
        uint64_t        lr_blkoff;      /* no longer used */
        blkptr_t        lr_blkptr;      /* spa block pointer for replay */
        /* write data will follow for small writes */
} lr_write_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_foid;        /* object id of file to truncate */
        uint64_t        lr_offset;      /* offset to truncate from */
        uint64_t        lr_length;      /* length to truncate */
} lr_truncate_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_foid;        /* file object to change attributes */
        uint64_t        lr_mask;        /* mask of attributes to set */
        uint64_t        lr_mode;        /* mode to set */
        uint64_t        lr_uid;         /* uid to set */
        uint64_t        lr_gid;         /* gid to set */
        uint64_t        lr_size;        /* size to set */
        uint64_t        lr_atime[2];    /* access time */
        uint64_t        lr_mtime[2];    /* modification time */
        /* optional attribute lr_attr_t may be here */
} lr_setattr_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_foid;        /* obj id of file */
        uint64_t        lr_aclcnt;      /* number of acl entries */
        /* lr_aclcnt number of ace_t entries follow this */
} lr_acl_v0_t;

typedef struct {
        lr_t            lr_common;      /* common portion of log record */
        uint64_t        lr_foid;        /* obj id of file */
        uint64_t        lr_aclcnt;      /* number of ACEs in ACL */
        uint64_t        lr_domcnt;      /* number of unique domains */
        uint64_t        lr_fuidcnt;     /* number of real fuids */
        uint64_t        lr_acl_bytes;   /* number of bytes in ACL */
        uint64_t        lr_acl_flags;   /* ACL flags */
        /* lr_acl_bytes number of variable sized ace’s follows */
} lr_acl_t;

/*
 * ZIL structure definitions, interface function prototype and globals.
 */

/*
 * Writes are handled in three different ways:
 *
 * WR_INDIRECT:
 *    In this mode, if we need to commit the write later, then the block
 *    is immediately written into the file system (using dmu_sync),
 *    and a pointer to the block is put into the log record.
 *    When the txg commits the block is linked in.
 *    This saves additionally writing the data into the log record.
 *    There are a few requirements for this to occur:
 *      - write is greater than zfs/zvol_immediate_write_sz
 *      - not using slogs (as slogs are assumed to always be faster
 *        than writing into the main pool)
 *      - the write occupies only one block
 * WR_COPIED:
 *    If we know we’ll immediately be committing the
 *    transaction (FSYNC or FDSYNC), the we allocate a larger
 *    log record here for the data and copy the data in.
 * WR_NEED_COPY:
 *    Otherwise we don’t allocate a buffer, and *if* we need to
 *    flush the write later then a buffer is allocated and
 *    we retrieve the data using the dmu.
 */
typedef enum {
        WR_INDIRECT,    /* indirect - a large write (dmu_sync() data */
                        /* and put blkptr in log, rather than actual data) */
        WR_COPIED,      /* immediate - data is copied into lr_write_t */
        WR_NEED_COPY,   /* immediate - data needs to be copied if pushed */
        WR_NUM_STATES   /* number of states */
} itx_wr_state_t;

typedef struct itx {
        list_node_t     itx_node;       /* linkage on zl_itx_list */
        void            *itx_private;   /* type-specific opaque data */
        itx_wr_state_t  itx_wr_state;   /* write state */
        uint8_t         itx_sync;       /* synchronous transaction */
        uint64_t        itx_sod;        /* record size on disk */
        uint64_t        itx_oid;        /* object id */
        lr_t            itx_lr;         /* common part of log record */
        /* followed by type-specific part of lr_xx_t and its immediate data */
} itx_t;

typedef int zil_parse_blk_func_t(zilog_t *zilog, blkptr_t *bp, void *arg,
    uint64_t txg);
typedef int zil_parse_lr_func_t(zilog_t *zilog, lr_t *lr, void *arg,
    uint64_t txg);
```

```
typedef int zil_replay_func_t();
typedef int zil_get_data_t(void *arg, lr_write_t *lr, char *dbuf, zio_t *zio);

extern int zil_parse(zilog_t *zilog, zil_parse_blk_func_t *parse_blk_func,
    zil_parse_lr_func_t *parse_lr_func, void *arg, uint64_t txg);

extern void     zil_init(void);
extern void     zil_fini(void);

extern zilog_t  *zil_alloc(objset_t *os, zil_header_t *zh_phys);
extern void     zil_free(zilog_t *zilog);

extern zilog_t  *zil_open(objset_t *os, zil_get_data_t *get_data);
extern void     zil_close(zilog_t *zilog);

extern void     zil_replay(objset_t *os, void *arg,
    zil_replay_func_t *replay_func[TX_MAX_TYPE]);
extern boolean_t zil_replaying(zilog_t *zilog, dmu_tx_t *tx);
extern void     zil_destroy(zilog_t *zilog, boolean_t keep_first);
extern void     zil_destroy_sync(zilog_t *zilog, dmu_tx_t *tx);
extern void     zil_rollback_destroy(zilog_t *zilog, dmu_tx_t *tx);

extern itx_t    *zil_itx_create(uint64_t txtype, size_t lrsize);
extern void     zil_itx_destroy(itx_t *itx);
extern void     zil_itx_assign(zilog_t *zilog, itx_t *itx, dmu_tx_t *tx);

extern void     zil_commit(zilog_t *zilog, uint64_t oid);

extern int      zil_vdev_offline(const char *osname, void *txarg);
extern int      zil_claim(struct dsl_pool *dp,
    struct dsl_dataset *ds, void *txarg);
extern int      zil_check_log_chain(struct dsl_pool *dp,
    struct dsl_dataset *ds, void *tx);
extern int      zil_claim(const char *osname, void *txarg);
extern int      zil_check_log_chain(const char *osname, void *txarg);
extern void     zil_sync(zilog_t *zilog, dmu_tx_t *tx);
extern void     zil_clean(zilog_t *zilog, uint64_t synced_txg);

extern int      zil_suspend(const char *osname, void **cookiep);
extern void     zil_resume(void *cookie);

extern void     zil_add_block(zilog_t *zilog, const blkptr_t *bp);
extern int      zil_bp_tree_add(zilog_t *zilog, const blkptr_t *bp);

extern void     zil_set_sync(zilog_t *zilog, uint64_t syncval);

extern void     zil_set_logbias(zilog_t *zilog, uint64_t slogval);

extern int zil_replay_disable;

#ifdef  __cplusplus
}
______unchanged_portion_omitted_
```

```
**********************************************************
   90344 Wed May  6 08:47:28 2015
new/usr/src/uts/common/fs/zfs/vdev.c
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
static int
vdev_count_leaves_impl(vdev_t *vd)
{
        int n = 0;

        if (vd->vdev_ops->vdev_op_leaf)
                return (1);

        for (int c = 0; c < vd->vdev_children; c++)
                n += vdev_count_leaves_impl(vd->vdev_child[c]);

        return (n);
}

int
vdev_count_leaves(spa_t *spa)
{
        return (vdev_count_leaves_impl(spa->spa_root_vdev));
}

#endif /* ! codereview */
void
vdev_add_child(vdev_t *pvd, vdev_t *cvd)
{
        size_t oldsize, newsize;
        uint64_t id = cvd->vdev_id;
        vdev_t **newchild;

        ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
        ASSERT(cvd->vdev_parent == NULL);

        cvd->vdev_parent = pvd;

        if (pvd == NULL)
                return;

        ASSERT(id >= pvd->vdev_children || pvd->vdev_child[id] == NULL);

        oldsize = pvd->vdev_children * sizeof (vdev_t *);
        pvd->vdev_children = MAX(pvd->vdev_children, id + 1);
        newsize = pvd->vdev_children * sizeof (vdev_t *);

        newchild = kmem_zalloc(newsize, KM_SLEEP);
        if (pvd->vdev_child != NULL) {
                bcopy(pvd->vdev_child, newchild, oldsize);
                kmem_free(pvd->vdev_child, oldsize);
        }

        pvd->vdev_child = newchild;
```

```
        pvd->vdev_child[id] = cvd;

        cvd->vdev_top = (pvd->vdev_top ? pvd->vdev_top: cvd);
        ASSERT(cvd->vdev_top->vdev_parent->vdev_parent == NULL);

        /*
         * Walk up all ancestors to update guid sum.
         */
        for (; pvd != NULL; pvd = pvd->vdev_parent)
                pvd->vdev_guid_sum += cvd->vdev_guid_sum;
}

void
vdev_remove_child(vdev_t *pvd, vdev_t *cvd)
{
        int c;
        uint_t id = cvd->vdev_id;

        ASSERT(cvd->vdev_parent == pvd);

        if (pvd == NULL)
                return;

        ASSERT(id < pvd->vdev_children);
        ASSERT(pvd->vdev_child[id] == cvd);

        pvd->vdev_child[id] = NULL;
        cvd->vdev_parent = NULL;

        for (c = 0; c < pvd->vdev_children; c++)
                if (pvd->vdev_child[c])
                        break;

        if (c == pvd->vdev_children) {
                kmem_free(pvd->vdev_child, c * sizeof (vdev_t *));
                pvd->vdev_child = NULL;
                pvd->vdev_children = 0;
        }

        /*
         * Walk up all ancestors to update guid sum.
         */
        for (; pvd != NULL; pvd = pvd->vdev_parent)
                pvd->vdev_guid_sum -= cvd->vdev_guid_sum;
}

/*
 * Remove any holes in the child array.
 */
void
vdev_compact_children(vdev_t *pvd)
{
        vdev_t **newchild, *cvd;
        int oldc = pvd->vdev_children;
        int newc;

        ASSERT(spa_config_held(pvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

        for (int c = newc = 0; c < oldc; c++)
                if (pvd->vdev_child[c])
                        newc++;

        newchild = kmem_alloc(newc * sizeof (vdev_t *), KM_SLEEP);

        for (int c = newc = 0; c < oldc; c++) {
                if ((cvd = pvd->vdev_child[c]) != NULL) {
                        newchild[newc] = cvd;
                        cvd->vdev_id = newc++;
                }
        }

        kmem_free(pvd->vdev_child, oldc * sizeof (vdev_t *));
        pvd->vdev_child = newchild;
        pvd->vdev_children = newc;
}

/*
 * Allocate and minimally initialize a vdev_t.
 */
vdev_t *
vdev_alloc_common(spa_t *spa, uint_t id, uint64_t guid, vdev_ops_t *ops)
{
        vdev_t *vd;

        vd = kmem_zalloc(sizeof (vdev_t), KM_SLEEP);

        if (spa->spa_root_vdev == NULL) {
                ASSERT(ops == &vdev_root_ops);
                spa->spa_root_vdev = vd;
                spa->spa_load_guid = spa_generate_guid(NULL);
        }

        if (guid == 0 && ops != &vdev_hole_ops) {
                if (spa->spa_root_vdev == vd) {
                        /*
                         * The root vdev’s guid will also be the pool guid,
                         * which must be unique among all pools.
                         */
                        guid = spa_generate_guid(NULL);
                } else {
                        /*
                         * Any other vdev’s guid must be unique within the pool.
                         */
                        guid = spa_generate_guid(spa);
                }
                ASSERT(!spa_guid_exists(spa_guid(spa), guid));
        }

        vd->vdev_spa = spa;
        vd->vdev_id = id;
        vd->vdev_guid = guid;
        vd->vdev_guid_sum = guid;
        vd->vdev_ops = ops;
        vd->vdev_state = VDEV_STATE_CLOSED;
        vd->vdev_ishole = (ops == &vdev_hole_ops);

        mutex_init(&vd->vdev_dtl_lock, NULL, MUTEX_DEFAULT, NULL);
        mutex_init(&vd->vdev_stat_lock, NULL, MUTEX_DEFAULT, NULL);
        mutex_init(&vd->vdev_probe_lock, NULL, MUTEX_DEFAULT, NULL);
        for (int t = 0; t < DTL_TYPES; t++) {
                vd->vdev_dtl[t] = range_tree_create(NULL, NULL,
                    &vd->vdev_dtl_lock);
        }
        txg_list_create(&vd->vdev_ms_list,
            offsetof(struct metaslab, ms_txg_node));
        txg_list_create(&vd->vdev_dtl_list,
            offsetof(struct vdev, vdev_dtl_node));
        vd->vdev_stat.vs_timestamp = gethrtime();
        vdev_queue_init(vd);
        vdev_cache_init(vd);

        return (vd);
```

```
}

/*
 * Allocate a new vdev.  The ’alloctype’ is used to control whether we are
 * creating a new vdev or loading an existing one - the behavior is slightly
 * different for each case.
 */
int
vdev_alloc(spa_t *spa, vdev_t **vdp, nvlist_t *nv, vdev_t *parent, uint_t id,
    int alloctype)
{
        vdev_ops_t *ops;
        char *type;
        uint64_t guid = 0, islog, nparity;
        vdev_t *vd;

        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_TYPE, &type) != 0)
                return (SET_ERROR(EINVAL));

        if ((ops = vdev_getops(type)) == NULL)
                return (SET_ERROR(EINVAL));

        /*
         * If this is a load, get the vdev guid from the nvlist.
         * Otherwise, vdev_alloc_common() will generate one for us.
         */
        if (alloctype == VDEV_ALLOC_LOAD) {
                uint64_t label_id;

                if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ID, &label_id) ||
                    label_id != id)
                        return (SET_ERROR(EINVAL));

                if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
                        return (SET_ERROR(EINVAL));
        } else if (alloctype == VDEV_ALLOC_SPARE) {
                if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
                        return (SET_ERROR(EINVAL));
        } else if (alloctype == VDEV_ALLOC_L2CACHE) {
                if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
                        return (SET_ERROR(EINVAL));
        } else if (alloctype == VDEV_ALLOC_ROOTPOOL) {
                if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
                        return (SET_ERROR(EINVAL));
        }

        /*
         * The first allocated vdev must be of type ’root’.
         */
        if (ops != &vdev_root_ops && spa->spa_root_vdev == NULL)
                return (SET_ERROR(EINVAL));

        /*
         * Determine whether we’re a log vdev.
         */
        islog = 0;
        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_LOG, &islog);
        if (islog && spa_version(spa) < SPA_VERSION_SLOGS)
                return (SET_ERROR(ENOTSUP));

        if (ops == &vdev_hole_ops && spa_version(spa) < SPA_VERSION_HOLES)
                return (SET_ERROR(ENOTSUP));

        /*
```

```
         * Set the nparity property for RAID-Z vdevs.
         */
        nparity = -1ULL;
        if (ops == &vdev_raidz_ops) {
                if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NPARITY,
                    &nparity) == 0) {
                        if (nparity == 0 || nparity > VDEV_RAIDZ_MAXPARITY)
                                return (SET_ERROR(EINVAL));
                        /*
                         * Previous versions could only support 1 or 2 parity
                         * device.
                         */
                        if (nparity > 1 &&
                            spa_version(spa) < SPA_VERSION_RAIDZ2)
                                return (SET_ERROR(ENOTSUP));
                        if (nparity > 2 &&
                            spa_version(spa) < SPA_VERSION_RAIDZ3)
                                return (SET_ERROR(ENOTSUP));
                } else {
                        /*
                         * We require the parity to be specified for SPAs that
                         * support multiple parity levels.
                         */
                        if (spa_version(spa) >= SPA_VERSION_RAIDZ2)
                                return (SET_ERROR(EINVAL));
                        /*
                         * Otherwise, we default to 1 parity device for RAID-Z.
                         */
                        nparity = 1;
                }
        } else {
                nparity = 0;
        }
        ASSERT(nparity != -1ULL);

        vd = vdev_alloc_common(spa, id, guid, ops);

        vd->vdev_islog = islog;
        vd->vdev_nparity = nparity;

        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PATH, &vd->vdev_path) == 0)
                vd->vdev_path = spa_strdup(vd->vdev_path);
        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_DEVID, &vd->vdev_devid) == 0)
                vd->vdev_devid = spa_strdup(vd->vdev_devid);
        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PHYS_PATH,
            &vd->vdev_physpath) == 0)
                vd->vdev_physpath = spa_strdup(vd->vdev_physpath);
        if (nvlist_lookup_string(nv, ZPOOL_CONFIG_FRU, &vd->vdev_fru) == 0)
                vd->vdev_fru = spa_strdup(vd->vdev_fru);

        /*
         * Set the whole_disk property.  If it's not specified, leave the value
         * as -1.
         */
        if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
            &vd->vdev_wholedisk) != 0)
                vd->vdev_wholedisk = -1ULL;

        /*
         * Look for the 'not present' flag.  This will only be set if the device
         * was not present at the time of import.
         */
        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT,
            &vd->vdev_not_present);

        /*
```

```
         * Get the alignment requirement.
         */
        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASHIFT, &vd->vdev_ashift);

        /*
         * Retrieve the vdev creation time.
         */
        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_CREATE_TXG,
            &vd->vdev_crtxg);

        /*
         * If we're a top-level vdev, try to load the allocation parameters.
         */
        if (parent && !parent->vdev_parent &&
            (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_SPLIT)) {
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
                    &vd->vdev_ms_array);
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
                    &vd->vdev_ms_shift);
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASIZE,
                    &vd->vdev_asize);
                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVING,
                    &vd->vdev_removing);
        }

        if (parent && !parent->vdev_parent && alloctype != VDEV_ALLOC_ATTACH) {
                ASSERT(alloctype == VDEV_ALLOC_LOAD ||
                    alloctype == VDEV_ALLOC_ADD ||
                    alloctype == VDEV_ALLOC_SPLIT ||
                    alloctype == VDEV_ALLOC_ROOTPOOL);
                vd->vdev_mg = metaslab_group_create(islog ?
                    spa_log_class(spa) : spa_normal_class(spa), vd);
        }

        /*
         * If we're a leaf vdev, try to load the DTL object and other state.
         */
        if (vd->vdev_ops->vdev_op_leaf &&
            (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_L2CACHE ||
            alloctype == VDEV_ALLOC_ROOTPOOL)) {
                if (alloctype == VDEV_ALLOC_LOAD) {
                        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DTL,
                            &vd->vdev_dtl_object);
                        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_UNSPARE,
                            &vd->vdev_unspare);
                }

                if (alloctype == VDEV_ALLOC_ROOTPOOL) {
                        uint64_t spare = 0;

                        if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_SPARE,
                            &spare) == 0 && spare)
                                spa_spare_add(vd);
                }

                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_OFFLINE,
                    &vd->vdev_offline);

                (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_RESILVER_TXG,
                    &vd->vdev_resilver_txg);

                /*
                 * When importing a pool, we want to ignore the persistent fault
                 * state, as the diagnosis made on another system may not be
                 * valid in the current context.  Local vdevs will
                 * remain in the faulted state.
                 */
                if (spa_load_state(spa) == SPA_LOAD_OPEN) {
                        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_FAULTED,
                            &vd->vdev_faulted);
                        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DEGRADED,
                            &vd->vdev_degraded);
                        (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVED,
                            &vd->vdev_removed);

                        if (vd->vdev_faulted || vd->vdev_degraded) {
                                char *aux;

                                vd->vdev_label_aux =
                                    VDEV_AUX_ERR_EXCEEDED;
                                if (nvlist_lookup_string(nv,
                                    ZPOOL_CONFIG_AUX_STATE, &aux) == 0 &&
                                    strcmp(aux, "external") == 0)
                                        vd->vdev_label_aux = VDEV_AUX_EXTERNAL;
                        }
                }
        }

        /*
         * Add ourselves to the parent's list of children.
         */
        vdev_add_child(parent, vd);

        *vdp = vd;

        return (0);
}

void
vdev_free(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;

        /*
         * vdev_free() implies closing the vdev first.  This is simpler than
         * trying to ensure complicated semantics for all callers.
         */
        vdev_close(vd);

        ASSERT(!list_link_active(&vd->vdev_config_dirty_node));
        ASSERT(!list_link_active(&vd->vdev_state_dirty_node));

        /*
         * Free all children.
         */
        for (int c = 0; c < vd->vdev_children; c++)
                vdev_free(vd->vdev_child[c]);

        ASSERT(vd->vdev_child == NULL);
        ASSERT(vd->vdev_guid_sum == vd->vdev_guid);

        /*
         * Discard allocation state.
         */
        if (vd->vdev_mg != NULL) {
                vdev_metaslab_fini(vd);
                metaslab_group_destroy(vd->vdev_mg);
        }

        ASSERT0(vd->vdev_stat.vs_space);
        ASSERT0(vd->vdev_stat.vs_dspace);
        ASSERT0(vd->vdev_stat.vs_alloc);
```

```

        /*
         * Remove this vdev from its parent's child list.
         */
        vdev_remove_child(vd->vdev_parent, vd);

        ASSERT(vd->vdev_parent == NULL);

        /*
         * Clean up vdev structure.
         */
        vdev_queue_fini(vd);
        vdev_cache_fini(vd);

        if (vd->vdev_path)
                spa_strfree(vd->vdev_path);
        if (vd->vdev_devid)
                spa_strfree(vd->vdev_devid);
        if (vd->vdev_physpath)
                spa_strfree(vd->vdev_physpath);
        if (vd->vdev_fru)
                spa_strfree(vd->vdev_fru);

        if (vd->vdev_isspare)
                spa_spare_remove(vd);
        if (vd->vdev_isl2cache)
                spa_l2cache_remove(vd);

        txg_list_destroy(&vd->vdev_ms_list);
        txg_list_destroy(&vd->vdev_dtl_list);

        mutex_enter(&vd->vdev_dtl_lock);
        space_map_close(vd->vdev_dtl_sm);
        for (int t = 0; t < DTL_TYPES; t++) {
                range_tree_vacate(vd->vdev_dtl[t], NULL, NULL);
                range_tree_destroy(vd->vdev_dtl[t]);
        }
        mutex_exit(&vd->vdev_dtl_lock);

        mutex_destroy(&vd->vdev_dtl_lock);
        mutex_destroy(&vd->vdev_stat_lock);
        mutex_destroy(&vd->vdev_probe_lock);

        if (vd == spa->spa_root_vdev)
                spa->spa_root_vdev = NULL;

        kmem_free(vd, sizeof (vdev_t));
}

/*
 * Transfer top-level vdev state from svd to tvd.
 */
static void
vdev_top_transfer(vdev_t *svd, vdev_t *tvd)
{
        spa_t *spa = svd->vdev_spa;
        metaslab_t *msp;
        vdev_t *vd;
        int t;

        ASSERT(tvd == tvd->vdev_top);

        tvd->vdev_ms_array = svd->vdev_ms_array;
        tvd->vdev_ms_shift = svd->vdev_ms_shift;
        tvd->vdev_ms_count = svd->vdev_ms_count;
        svd->vdev_ms_array = 0;
        svd->vdev_ms_shift = 0;
        svd->vdev_ms_count = 0;

        if (tvd->vdev_mg)
                ASSERT3P(tvd->vdev_mg, ==, svd->vdev_mg);
        tvd->vdev_mg = svd->vdev_mg;
        tvd->vdev_ms = svd->vdev_ms;

        svd->vdev_mg = NULL;
        svd->vdev_ms = NULL;

        if (tvd->vdev_mg != NULL)
                tvd->vdev_mg->mg_vd = tvd;

        tvd->vdev_stat.vs_alloc = svd->vdev_stat.vs_alloc;
        tvd->vdev_stat.vs_space = svd->vdev_stat.vs_space;
        tvd->vdev_stat.vs_dspace = svd->vdev_stat.vs_dspace;

        svd->vdev_stat.vs_alloc = 0;
        svd->vdev_stat.vs_space = 0;
        svd->vdev_stat.vs_dspace = 0;

        for (t = 0; t < TXG_SIZE; t++) {
                while ((msp = txg_list_remove(&svd->vdev_ms_list, t)) != NULL)
                        (void) txg_list_add(&tvd->vdev_ms_list, msp, t);
                while ((vd = txg_list_remove(&svd->vdev_dtl_list, t)) != NULL)
                        (void) txg_list_add(&tvd->vdev_dtl_list, vd, t);
                if (txg_list_remove_this(&spa->spa_vdev_txg_list, svd, t))
                        (void) txg_list_add(&spa->spa_vdev_txg_list, tvd, t);
        }

        if (list_link_active(&svd->vdev_config_dirty_node)) {
                vdev_config_clean(svd);
                vdev_config_dirty(tvd);
        }

        if (list_link_active(&svd->vdev_state_dirty_node)) {
                vdev_state_clean(svd);
                vdev_state_dirty(tvd);
        }

        tvd->vdev_deflate_ratio = svd->vdev_deflate_ratio;
        svd->vdev_deflate_ratio = 0;

        tvd->vdev_islog = svd->vdev_islog;
        svd->vdev_islog = 0;
}

static void
vdev_top_update(vdev_t *tvd, vdev_t *vd)
{
        if (vd == NULL)
                return;

        vd->vdev_top = tvd;

        for (int c = 0; c < vd->vdev_children; c++)
                vdev_top_update(tvd, vd->vdev_child[c]);
}

/*
 * Add a mirror/replacing vdev above an existing vdev.
 */
vdev_t *
vdev_add_parent(vdev_t *cvd, vdev_ops_t *ops)
```

```
{
        spa_t *spa = cvd->vdev_spa;
        vdev_t *pvd = cvd->vdev_parent;
        vdev_t *mvd;

        ASSERT(spa_config_held(spa, SCL_ALL, RW_WRITER) == SCL_ALL);

        mvd = vdev_alloc_common(spa, cvd->vdev_id, 0, ops);

        mvd->vdev_asize = cvd->vdev_asize;
        mvd->vdev_min_asize = cvd->vdev_min_asize;
        mvd->vdev_max_asize = cvd->vdev_max_asize;
        mvd->vdev_ashift = cvd->vdev_ashift;
        mvd->vdev_state = cvd->vdev_state;
        mvd->vdev_crtxg = cvd->vdev_crtxg;

        vdev_remove_child(pvd, cvd);
        vdev_add_child(pvd, mvd);
        cvd->vdev_id = mvd->vdev_children;
        vdev_add_child(mvd, cvd);
        vdev_top_update(cvd->vdev_top, cvd->vdev_top);

        if (mvd == mvd->vdev_top)
                vdev_top_transfer(cvd, mvd);

        return (mvd);
}

/*
 * Remove a 1-way mirror/replacing vdev from the tree.
 */
void
vdev_remove_parent(vdev_t *cvd)
{
        vdev_t *mvd = cvd->vdev_parent;
        vdev_t *pvd = mvd->vdev_parent;

        ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

        ASSERT(mvd->vdev_children == 1);
        ASSERT(mvd->vdev_ops == &vdev_mirror_ops ||
            mvd->vdev_ops == &vdev_replacing_ops ||
            mvd->vdev_ops == &vdev_spare_ops);
        cvd->vdev_ashift = mvd->vdev_ashift;

        vdev_remove_child(mvd, cvd);
        vdev_remove_child(pvd, mvd);

        /*
         * If cvd will replace mvd as a top-level vdev, preserve mvd's guid.
         * Otherwise, we could have detached an offline device, and when we
         * go to import the pool we'll think we have two top-level vdevs,
         * instead of a different version of the same top-level vdev.
         */
        if (mvd->vdev_top == mvd) {
                uint64_t guid_delta = mvd->vdev_guid - cvd->vdev_guid;
                cvd->vdev_orig_guid = cvd->vdev_guid;
                cvd->vdev_guid += guid_delta;
                cvd->vdev_guid_sum += guid_delta;
        }
        cvd->vdev_id = mvd->vdev_id;
        vdev_add_child(pvd, cvd);
        vdev_top_update(cvd->vdev_top, cvd->vdev_top);

        if (cvd == cvd->vdev_top)
                vdev_top_transfer(mvd, cvd);
        ASSERT(mvd->vdev_children == 0);
        vdev_free(mvd);
}

int
vdev_metaslab_init(vdev_t *vd, uint64_t txg)
{
        spa_t *spa = vd->vdev_spa;
        objset_t *mos = spa->spa_meta_objset;
        uint64_t m;
        uint64_t oldc = vd->vdev_ms_count;
        uint64_t newc = vd->vdev_asize >> vd->vdev_ms_shift;
        metaslab_t **mspp;
        int error;

        ASSERT(txg == 0 || spa_config_held(spa, SCL_ALLOC, RW_WRITER));

        /*
         * This vdev is not being allocated from yet or is a hole.
         */
        if (vd->vdev_ms_shift == 0)
                return (0);

        ASSERT(!vd->vdev_ishole);

        /*
         * Compute the raidz-deflation ratio.  Note, we hard-code
         * in 128k (1 << 17) because it is the "typical" blocksize.
         * Even though SPA_MAXBLOCKSIZE changed, this algorithm can not change,
         * otherwise it would inconsistently account for existing bp’s.
         */
        vd->vdev_deflate_ratio = (1 << 17) /
            (vdev_psize_to_asize(vd, 1 << 17) >> SPA_MINBLOCKSHIFT);

        ASSERT(oldc <= newc);

        mspp = kmem_zalloc(newc * sizeof (*mspp), KM_SLEEP);

        if (oldc != 0) {
                bcopy(vd->vdev_ms, mspp, oldc * sizeof (*mspp));
                kmem_free(vd->vdev_ms, oldc * sizeof (*mspp));
        }

        vd->vdev_ms = mspp;
        vd->vdev_ms_count = newc;

        for (m = oldc; m < newc; m++) {
                uint64_t object = 0;

                if (txg == 0) {
                        error = dmu_read(mos, vd->vdev_ms_array,
                            m * sizeof (uint64_t), sizeof (uint64_t), &object,
                            DMU_READ_PREFETCH);
                        if (error)
                                return (error);
                }

                error = metaslab_init(vd->vdev_mg, m, object, txg,
                    &(vd->vdev_ms[m]));
                if (error)
                        return (error);
        }

        if (txg == 0)
                spa_config_enter(spa, SCL_ALLOC, FTAG, RW_WRITER);
```

```
        /*
         * If the vdev is being removed we don’t activate
         * the metaslabs since we want to ensure that no new
         * allocations are performed on this device.
         */
        if (oldc == 0 && !vd->vdev_removing)
                metaslab_group_activate(vd->vdev_mg);

        if (txg == 0)
                spa_config_exit(spa, SCL_ALLOC, FTAG);

        return (0);
}

void
vdev_metaslab_fini(vdev_t *vd)
{
        uint64_t m;
        uint64_t count = vd->vdev_ms_count;

        if (vd->vdev_ms != NULL) {
                metaslab_group_passivate(vd->vdev_mg);
                for (m = 0; m < count; m++) {
                        metaslab_t *msp = vd->vdev_ms[m];

                        if (msp != NULL)
                                metaslab_fini(msp);
                }
                kmem_free(vd->vdev_ms, count * sizeof (metaslab_t *));
                vd->vdev_ms = NULL;
        }
}

typedef struct vdev_probe_stats {
        boolean_t       vps_readable;
        boolean_t       vps_writeable;
        int             vps_flags;
} vdev_probe_stats_t;

static void
vdev_probe_done(zio_t *zio)
{
        spa_t *spa = zio->io_spa;
        vdev_t *vd = zio->io_vd;
        vdev_probe_stats_t *vps = zio->io_private;

        ASSERT(vd->vdev_probe_zio != NULL);

        if (zio->io_type == ZIO_TYPE_READ) {
                if (zio->io_error == 0)
                        vps->vps_readable = 1;
                if (zio->io_error == 0 && spa_writeable(spa)) {
                        zio_nowait(zio_write_phys(vd->vdev_probe_zio, vd,
                            zio->io_offset, zio->io_size, zio->io_data,
                            ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
                            ZIO_PRIORITY_SYNC_WRITE, vps->vps_flags, B_TRUE));
                } else {
                        zio_buf_free(zio->io_data, zio->io_size);
                }
        } else if (zio->io_type == ZIO_TYPE_WRITE) {
                if (zio->io_error == 0)
                        vps->vps_writeable = 1;
                zio_buf_free(zio->io_data, zio->io_size);
        } else if (zio->io_type == ZIO_TYPE_NULL) {
                zio_t *pio;
                vd->vdev_cant_read |= !vps->vps_readable;
                vd->vdev_cant_write |= !vps->vps_writeable;

                if (vdev_readable(vd) &&
                    (vdev_writeable(vd) || !spa_writeable(spa))) {
                        zio->io_error = 0;
                } else {
                        ASSERT(zio->io_error != 0);
                        zfs_ereport_post(FM_EREPORT_ZFS_PROBE_FAILURE,
                            spa, vd, NULL, 0, 0);
                        zio->io_error = SET_ERROR(ENXIO);
                }

                mutex_enter(&vd->vdev_probe_lock);
                ASSERT(vd->vdev_probe_zio == zio);
                vd->vdev_probe_zio = NULL;
                mutex_exit(&vd->vdev_probe_lock);

                while ((pio = zio_walk_parents(zio)) != NULL)
                        if (!vdev_accessible(vd, pio))
                                pio->io_error = SET_ERROR(ENXIO);

                kmem_free(vps, sizeof (*vps));
        }
}

/*
 * Determine whether this device is accessible.
 *
 * Read and write to several known locations: the pad regions of each
 * vdev label but the first, which we leave alone in case it contains
 * a VTOC.
 */
zio_t *
vdev_probe(vdev_t *vd, zio_t *zio)
{
        spa_t *spa = vd->vdev_spa;
        vdev_probe_stats_t *vps = NULL;
        zio_t *pio;

        ASSERT(vd->vdev_ops->vdev_op_leaf);

        /*
         * Don't probe the probe.
         */
        if (zio && (zio->io_flags & ZIO_FLAG_PROBE))
                return (NULL);

        /*
         * To prevent 'probe storms' when a device fails, we create
         * just one probe i/o at a time.  All zios that want to probe
         * this vdev will become parents of the probe io.
         */
        mutex_enter(&vd->vdev_probe_lock);

        if ((pio = vd->vdev_probe_zio) == NULL) {
                vps = kmem_zalloc(sizeof (*vps), KM_SLEEP);

                vps->vps_flags = ZIO_FLAG_CANFAIL | ZIO_FLAG_PROBE |
                    ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE |
                    ZIO_FLAG_TRYHARD;

                if (spa_config_held(spa, SCL_ZIO, RW_WRITER)) {
                        /*
                         * vdev_cant_read and vdev_cant_write can only
```

```
                         * transition from TRUE to FALSE when we have the
                         * SCL_ZIO lock as writer; otherwise they can only
                         * transition from FALSE to TRUE.  This ensures that
                         * any zio looking at these values can assume that
                         * failures persist for the life of the I/O.  That's
                         * important because when a device has intermittent
                         * connectivity problems, we want to ensure that
                         * they're ascribed to the device (ENXIO) and not
                         * the zio (EIO).
                         *
                         * Since we hold SCL_ZIO as writer here, clear both
                         * values so the probe can reevaluate from first
                         * principles.
                         */
                        vps->vps_flags |= ZIO_FLAG_CONFIG_WRITER;
                        vd->vdev_cant_read = B_FALSE;
                        vd->vdev_cant_write = B_FALSE;
                }

                vd->vdev_probe_zio = pio = zio_null(NULL, spa, vd,
                    vdev_probe_done, vps,
                    vps->vps_flags | ZIO_FLAG_DONT_PROPAGATE);

                /*
                 * We can't change the vdev state in this context, so we
                 * kick off an async task to do it on our behalf.
                 */
                if (zio != NULL) {
                        vd->vdev_probe_wanted = B_TRUE;
                        spa_async_request(spa, SPA_ASYNC_PROBE);
                }
        }

        if (zio != NULL)
                zio_add_child(zio, pio);

        mutex_exit(&vd->vdev_probe_lock);

        if (vps == NULL) {
                ASSERT(zio != NULL);
                return (NULL);
        }

        for (int l = 1; l < VDEV_LABELS; l++) {
                zio_nowait(zio_read_phys(pio, vd,
                    vdev_label_offset(vd->vdev_psize, l,
                    offsetof(vdev_label_t, vl_pad2)),
                    VDEV_PAD_SIZE, zio_buf_alloc(VDEV_PAD_SIZE),
                    ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
                    ZIO_PRIORITY_SYNC_READ, vps->vps_flags, B_TRUE));
        }

        if (zio == NULL)
                return (pio);

        zio_nowait(pio);
        return (NULL);
}

static void
vdev_open_child(void *arg)
{
        vdev_t *vd = arg;

        vd->vdev_open_thread = curthread;
        vd->vdev_open_error = vdev_open(vd);
        vd->vdev_open_thread = NULL;
}

boolean_t
vdev_uses_zvols(vdev_t *vd)
{
        if (vd->vdev_path && strncmp(vd->vdev_path, ZVOL_DIR,
            strlen(ZVOL_DIR)) == 0)
                return (B_TRUE);
        for (int c = 0; c < vd->vdev_children; c++)
                if (vdev_uses_zvols(vd->vdev_child[c]))
                        return (B_TRUE);
        return (B_FALSE);
}

void
vdev_open_children(vdev_t *vd)
{
        taskq_t *tq;
        int children = vd->vdev_children;

        /*
         * in order to handle pools on top of zvols, do the opens
         * in a single thread so that the same thread holds the
         * spa_namespace_lock
         */
        if (vdev_uses_zvols(vd)) {
                for (int c = 0; c < children; c++)
                        vd->vdev_child[c]->vdev_open_error =
                            vdev_open(vd->vdev_child[c]);
                return;
        }
        tq = taskq_create("vdev_open", children, minclsyspri,
            children, children, TASKQ_PREPOPULATE);

        for (int c = 0; c < children; c++)
                VERIFY(taskq_dispatch(tq, vdev_open_child, vd->vdev_child[c],
                    TQ_SLEEP) != NULL);

        taskq_destroy(tq);
}

/*
 * Prepare a virtual device for access.
 */
int
vdev_open(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;
        int error;
        uint64_t osize = 0;
        uint64_t max_osize = 0;
        uint64_t asize, max_asize, psize;
        uint64_t ashift = 0;

        ASSERT(vd->vdev_open_thread == curthread ||
            spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
        ASSERT(vd->vdev_state == VDEV_STATE_CLOSED ||
            vd->vdev_state == VDEV_STATE_CANT_OPEN ||
            vd->vdev_state == VDEV_STATE_OFFLINE);

        vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
        vd->vdev_cant_read = B_FALSE;
        vd->vdev_cant_write = B_FALSE;
        vd->vdev_min_asize = vdev_get_min_asize(vd);
```

```
        /*
         * If this vdev is not removed, check its fault status.  If it’s
         * faulted, bail out of the open.
         */
        if (!vd->vdev_removed && vd->vdev_faulted) {
                ASSERT(vd->vdev_children == 0);
                ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
                    vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
                vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
                    vd->vdev_label_aux);
                return (SET_ERROR(ENXIO));
        } else if (vd->vdev_offline) {
                ASSERT(vd->vdev_children == 0);
                vdev_set_state(vd, B_TRUE, VDEV_STATE_OFFLINE, VDEV_AUX_NONE);
                return (SET_ERROR(ENXIO));
        }

        error = vd->vdev_ops->vdev_op_open(vd, &osize, &max_osize, &ashift);

        /*
         * Reset the vdev_reopening flag so that we actually close
         * the vdev on error.
         */
        vd->vdev_reopening = B_FALSE;
        if (zio_injection_enabled && error == 0)
                error = zio_handle_device_injection(vd, NULL, ENXIO);

        if (error) {
                if (vd->vdev_removed &&
                    vd->vdev_stat.vs_aux != VDEV_AUX_OPEN_FAILED)
                        vd->vdev_removed = B_FALSE;

                vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
                    vd->vdev_stat.vs_aux);
                return (error);
        }

        vd->vdev_removed = B_FALSE;

        /*
         * Recheck the faulted flag now that we have confirmed that
         * the vdev is accessible.  If we’re faulted, bail.
         */
        if (vd->vdev_faulted) {
                ASSERT(vd->vdev_children == 0);
                ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
                    vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
                vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
                    vd->vdev_label_aux);
                return (SET_ERROR(ENXIO));
        }

        if (vd->vdev_degraded) {
                ASSERT(vd->vdev_children == 0);
                vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
                    VDEV_AUX_ERR_EXCEEDED);
        } else {
                vdev_set_state(vd, B_TRUE, VDEV_STATE_HEALTHY, 0);
        }

        /*
         * For hole or missing vdevs we just return success.
         */
        if (vd->vdev_ishole || vd->vdev_ops == &vdev_missing_ops)
                return (0);
        for (int c = 0; c < vd->vdev_children; c++) {
                if (vd->vdev_child[c]->vdev_state != VDEV_STATE_HEALTHY) {
                        vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
                            VDEV_AUX_NONE);
                        break;
                }
        }

        osize = P2ALIGN(osize, (uint64_t)sizeof (vdev_label_t));
        max_osize = P2ALIGN(max_osize, (uint64_t)sizeof (vdev_label_t));

        if (vd->vdev_children == 0) {
                if (osize < SPA_MINDEVSIZE) {
                        vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_TOO_SMALL);
                        return (SET_ERROR(EOVERFLOW));
                }
                psize = osize;
                asize = osize - (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE);
                max_asize = max_osize - (VDEV_LABEL_START_SIZE +
                    VDEV_LABEL_END_SIZE);
        } else {
                if (vd->vdev_parent != NULL && osize < SPA_MINDEVSIZE -
                    (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE)) {
                        vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_TOO_SMALL);
                        return (SET_ERROR(EOVERFLOW));
                }
                psize = 0;
                asize = osize;
                max_asize = max_osize;
        }

        vd->vdev_psize = psize;

        /*
         * Make sure the allocatable size hasn’t shrunk.
         */
        if (asize < vd->vdev_min_asize) {
                vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
                    VDEV_AUX_BAD_LABEL);
                return (SET_ERROR(EINVAL));
        }

        if (vd->vdev_asize == 0) {
                /*
                 * This is the first-ever open, so use the computed values.
                 * For testing purposes, a higher ashift can be requested.
                 */
                vd->vdev_asize = asize;
                vd->vdev_max_asize = max_asize;
                vd->vdev_ashift = MAX(ashift, vd->vdev_ashift);
        } else {
                /*
                 * Detect if the alignment requirement has increased.
                 * We don’t want to make the pool unavailable, just
                 * issue a warning instead.
                 */
                if (ashift > vd->vdev_top->vdev_ashift &&
                    vd->vdev_ops->vdev_op_leaf) {
                        cmn_err(CE_WARN,
                            "Disk, ’%s’, has a block alignment that is "
                            "larger than the pool’s alignment\n",
                            vd->vdev_path);
                }
                vd->vdev_max_asize = max_asize;
```

```
        }

        /*
         * If all children are healthy and the asize has increased,
         * then we’ve experienced dynamic LUN growth.  If automatic
         * expansion is enabled then use the additional space.
         */
        if (vd->vdev_state == VDEV_STATE_HEALTHY && asize > vd->vdev_asize &&
            (vd->vdev_expanding || spa->spa_autoexpand))
                vd->vdev_asize = asize;

        vdev_set_min_asize(vd);

        /*
         * Ensure we can issue some IO before declaring the
         * vdev open for business.
         */
        if (vd->vdev_ops->vdev_op_leaf &&
            (error = zio_wait(vdev_probe(vd, NULL))) != 0) {
                vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
                    VDEV_AUX_ERR_EXCEEDED);
                return (error);
        }

        /*
         * If a leaf vdev has a DTL, and seems healthy, then kick off a
         * resilver.  But don’t do this if we are doing a reopen for a scrub,
         * since this would just restart the scrub we are already doing.
         */
        if (vd->vdev_ops->vdev_op_leaf && !spa->spa_scrub_reopen &&
            vdev_resilver_needed(vd, NULL, NULL))
                spa_async_request(spa, SPA_ASYNC_RESILVER);

        return (0);
}

/*
 * Called once the vdevs are all opened, this routine validates the label
 * contents.  This needs to be done before vdev_load() so that we don’t
 * inadvertently do repair I/Os to the wrong device.
 *
 * If ’strict’ is false ignore the spa guid check. This is necessary because
 * if the machine crashed during a re-guid the new guid might have been written
 * to all of the vdev labels, but not the cached config. The strict check
 * will be performed when the pool is opened again using the mos config.
 *
 * This function will only return failure if one of the vdevs indicates that it
 * has since been destroyed or exported.  This is only possible if
 * /etc/zfs/zpool.cache was readonly at the time.  Otherwise, the vdev state
 * will be updated but the function will return 0.
 */
int
vdev_validate(vdev_t *vd, boolean_t strict)
{
        spa_t *spa = vd->vdev_spa;
        nvlist_t *label;
        uint64_t guid = 0, top_guid;
        uint64_t state;

        for (int c = 0; c < vd->vdev_children; c++)
                if (vdev_validate(vd->vdev_child[c], strict) != 0)
                        return (SET_ERROR(EBADF));

        /*
         * If the device has already failed, or was marked offline, don’t do
         * any further validation.  Otherwise, label I/O will fail and we will
         * overwrite the previous state.
         */
        if (vd->vdev_ops->vdev_op_leaf && vdev_readable(vd)) {
                uint64_t aux_guid = 0;
                nvlist_t *nvl;
                uint64_t txg = spa_last_synced_txg(spa) != 0 ?
                    spa_last_synced_txg(spa) : -1ULL;

                if ((label = vdev_label_read_config(vd, txg)) == NULL) {
                        vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_BAD_LABEL);
                        return (0);
                }

                /*
                 * Determine if this vdev has been split off into another
                 * pool.  If so, then refuse to open it.
                 */
                if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_SPLIT_GUID,
                    &aux_guid) == 0 && aux_guid == spa_guid(spa)) {
                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_SPLIT_POOL);
                        nvlist_free(label);
                        return (0);
                }

                if (strict && (nvlist_lookup_uint64(label,
                    ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
                    guid != spa_guid(spa))) {
                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_CORRUPT_DATA);
                        nvlist_free(label);
                        return (0);
                }

                if (nvlist_lookup_nvlist(label, ZPOOL_CONFIG_VDEV_TREE, &nvl)
                    != 0 || nvlist_lookup_uint64(nvl, ZPOOL_CONFIG_ORIG_GUID,
                    &aux_guid) != 0)
                        aux_guid = 0;

                /*
                 * If this vdev just became a top-level vdev because its
                 * sibling was detached, it will have adopted the parent’s
                 * vdev guid -- but the label may or may not be on disk yet.
                 * Fortunately, either version of the label will have the
                 * same top guid, so if we’re a top-level vdev, we can
                 * safely compare to that instead.
                 *
                 * If we split this vdev off instead, then we also check the
                 * original pool’s guid.  We don’t want to consider the vdev
                 * corrupt if it is partway through a split operation.
                 */
                if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID,
                    &guid) != 0 ||
                    nvlist_lookup_uint64(label, ZPOOL_CONFIG_TOP_GUID,
                    &top_guid) != 0 ||
                    ((vd->vdev_guid != guid && vd->vdev_guid != aux_guid) &&
                    (vd->vdev_guid != top_guid || vd != vd->vdev_top))) {
                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_CORRUPT_DATA);
                        nvlist_free(label);
                        return (0);
                }

                if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE,
                    &state) != 0) {
```

```
                        vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_CORRUPT_DATA);
                        nvlist_free(label);
                        return (0);
                }

                nvlist_free(label);

                /*
                 * If this is a verbatim import, no need to check the
                 * state of the pool.
                 */
                if (!(spa->spa_import_flags & ZFS_IMPORT_VERBATIM) &&
                    spa_load_state(spa) == SPA_LOAD_OPEN &&
                    state != POOL_STATE_ACTIVE)
                        return (SET_ERROR(EBADF));

                /*
                 * If we were able to open and validate a vdev that was
                 * previously marked permanently unavailable, clear that state
                 * now.
                 */
                if (vd->vdev_not_present)
                        vd->vdev_not_present = 0;
        }

        return (0);
}

/*
 * Close a virtual device.
 */
void
vdev_close(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;
        vdev_t *pvd = vd->vdev_parent;

        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

        /*
         * If our parent is reopening, then we are as well, unless we are
         * going offline.
         */
        if (pvd != NULL && pvd->vdev_reopening)
                vd->vdev_reopening = (pvd->vdev_reopening && !vd->vdev_offline);

        vd->vdev_ops->vdev_op_close(vd);

        vdev_cache_purge(vd);

        /*
         * We record the previous state before we close it, so that if we are
         * doing a reopen(), we don’t generate FMA ereports if we notice that
         * it’s still faulted.
         */
        vd->vdev_prevstate = vd->vdev_state;

        if (vd->vdev_offline)
                vd->vdev_state = VDEV_STATE_OFFLINE;
        else
                vd->vdev_state = VDEV_STATE_CLOSED;
        vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
}

void
```

```
vdev_hold(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;

        ASSERT(spa_is_root(spa));
        if (spa->spa_state == POOL_STATE_UNINITIALIZED)
                return;

        for (int c = 0; c < vd->vdev_children; c++)
                vdev_hold(vd->vdev_child[c]);

        if (vd->vdev_ops->vdev_op_leaf)
                vd->vdev_ops->vdev_op_hold(vd);
}

void
vdev_rele(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;

        ASSERT(spa_is_root(spa));
        for (int c = 0; c < vd->vdev_children; c++)
                vdev_rele(vd->vdev_child[c]);

        if (vd->vdev_ops->vdev_op_leaf)
                vd->vdev_ops->vdev_op_rele(vd);
}

/*
 * Reopen all interior vdevs and any unopened leaves.  We don't actually
 * reopen leaf vdevs which had previously been opened as they might deadlock
 * on the spa_config_lock.  Instead we only obtain the leaf's physical size.
 * If the leaf has never been opened then open it, as usual.
 */
void
vdev_reopen(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;

        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

        /* set the reopening flag unless we're taking the vdev offline */
        vd->vdev_reopening = !vd->vdev_offline;
        vdev_close(vd);
        (void) vdev_open(vd);

        /*
         * Call vdev_validate() here to make sure we have the same device.
         * Otherwise, a device with an invalid label could be successfully
         * opened in response to vdev_reopen().
         */
        if (vd->vdev_aux) {
                (void) vdev_validate_aux(vd);
                if (vdev_readable(vd) && vdev_writeable(vd) &&
                    vd->vdev_aux == &spa->spa_l2cache &&
                    !l2arc_vdev_present(vd))
                        l2arc_add_vdev(spa, vd);
        } else {
                (void) vdev_validate(vd, B_TRUE);
        }

        /*
         * Reassess parent vdev's health.
         */
        vdev_propagate_state(vd);
}
```

```
int
vdev_create(vdev_t *vd, uint64_t txg, boolean_t isreplacing)
{
        int error;

        /*
         * Normally, partial opens (e.g. of a mirror) are allowed.
         * For a create, however, we want to fail the request if
         * there are any components we can't open.
         */
        error = vdev_open(vd);

        if (error || vd->vdev_state != VDEV_STATE_HEALTHY) {
                vdev_close(vd);
                return (error ? error : ENXIO);
        }

        /*
         * Recursively load DTLs and initialize all labels.
         */
        if ((error = vdev_dtl_load(vd)) != 0 ||
            (error = vdev_label_init(vd, txg, isreplacing ?
            VDEV_LABEL_REPLACE : VDEV_LABEL_CREATE)) != 0) {
                vdev_close(vd);
                return (error);
        }

        return (0);
}

void
vdev_metaslab_set_size(vdev_t *vd)
{
        /*
         * Aim for roughly metaslabs_per_vdev (default 200) metaslabs per vdev.
         */
        vd->vdev_ms_shift = highbit64(vd->vdev_asize / metaslabs_per_vdev);
        vd->vdev_ms_shift = MAX(vd->vdev_ms_shift, SPA_MAXBLOCKSHIFT);
}

void
vdev_dirty(vdev_t *vd, int flags, void *arg, uint64_t txg)
{
        ASSERT(vd == vd->vdev_top);
        ASSERT(!vd->vdev_ishole);
        ASSERT(ISP2(flags));
        ASSERT(spa_writeable(vd->vdev_spa));

        if (flags & VDD_METASLAB)
                (void) txg_list_add(&vd->vdev_ms_list, arg, txg);

        if (flags & VDD_DTL)
                (void) txg_list_add(&vd->vdev_dtl_list, arg, txg);

        (void) txg_list_add(&vd->vdev_spa->spa_vdev_txg_list, vd, txg);
}

void
vdev_dirty_leaves(vdev_t *vd, int flags, uint64_t txg)
{
        for (int c = 0; c < vd->vdev_children; c++)
                vdev_dirty_leaves(vd->vdev_child[c], flags, txg);

        if (vd->vdev_ops->vdev_op_leaf)
                vdev_dirty(vd->vdev_top, flags, vd, txg);
```

```
}

/*
 * DTLs.
 *
 * A vdev's DTL (dirty time log) is the set of transaction groups for which
 * the vdev has less than perfect replication.  There are four kinds of DTL:
 *
 * DTL_MISSING: txgs for which the vdev has no valid copies of the data
 *
 * DTL_PARTIAL: txgs for which data is available, but not fully replicated
 *
 * DTL_SCRUB: the txgs that could not be repaired by the last scrub; upon
 *      scrub completion, DTL_SCRUB replaces DTL_MISSING in the range of
 *      txgs that was scrubbed.
 *
 * DTL_OUTAGE: txgs which cannot currently be read, whether due to
 *      persistent errors or just some device being offline.
 *      Unlike the other three, the DTL_OUTAGE map is not generally
 *      maintained; it's only computed when needed, typically to
 *      determine whether a device can be detached.
 *
 * For leaf vdevs, DTL_MISSING and DTL_PARTIAL are identical: the device
 * either has the data or it doesn't.
 *
 * For interior vdevs such as mirror and RAID-Z the picture is more complex.
 * A vdev's DTL_PARTIAL is the union of its children's DTL_PARTIALs, because
 * if any child is less than fully replicated, then so is its parent.
 * A vdev's DTL_MISSING is a modified union of its children's DTL_MISSINGs,
 * comprising only those txgs which appear in 'maxfaults' or more children;
 * those are the txgs we don't have enough replication to read.  For example,
 * double-parity RAID-Z can tolerate up to two missing devices (maxfaults == 2);
 * thus, its DTL_MISSING consists of the set of txgs that appear in more than
 * two child DTL_MISSING maps.
 *
 * It should be clear from the above that to compute the DTLs and outage maps
 * for all vdevs, it suffices to know just the leaf vdevs' DTL_MISSING maps.
 * Therefore, that is all we keep on disk.  When loading the pool, or after
 * a configuration change, we generate all other DTLs from first principles.
 */
void
vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
{
        range_tree_t *rt = vd->vdev_dtl[t];

        ASSERT(t < DTL_TYPES);
        ASSERT(vd != vd->vdev_spa->spa_root_vdev);
        ASSERT(spa_writeable(vd->vdev_spa));

        mutex_enter(rt->rt_lock);
        if (!range_tree_contains(rt, txg, size))
                range_tree_add(rt, txg, size);
        mutex_exit(rt->rt_lock);
}

boolean_t
vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
{
        range_tree_t *rt = vd->vdev_dtl[t];
        boolean_t dirty = B_FALSE;

        ASSERT(t < DTL_TYPES);
        ASSERT(vd != vd->vdev_spa->spa_root_vdev);

        mutex_enter(rt->rt_lock);
        if (range_tree_space(rt) != 0)
```

```
                dirty = range_tree_contains(rt, txg, size);
        mutex_exit(rt->rt_lock);

        return (dirty);
}

boolean_t
vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t t)
{
        range_tree_t *rt = vd->vdev_dtl[t];
        boolean_t empty;

        mutex_enter(rt->rt_lock);
        empty = (range_tree_space(rt) == 0);
        mutex_exit(rt->rt_lock);

        return (empty);
}

/*
 * Returns the lowest txg in the DTL range.
 */
static uint64_t
vdev_dtl_min(vdev_t *vd)
{
        range_seg_t *rs;

        ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
        ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
        ASSERT0(vd->vdev_children);

        rs = avl_first(&vd->vdev_dtl[DTL_MISSING]->rt_root);
        return (rs->rs_start - 1);
}

/*
 * Returns the highest txg in the DTL.
 */
static uint64_t
vdev_dtl_max(vdev_t *vd)
{
        range_seg_t *rs;

        ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
        ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
        ASSERT0(vd->vdev_children);

        rs = avl_last(&vd->vdev_dtl[DTL_MISSING]->rt_root);
        return (rs->rs_end);
}

/*
 * Determine if a resilvering vdev should remove any DTL entries from
 * its range. If the vdev was resilvering for the entire duration of the
 * scan then it should excise that range from its DTLs. Otherwise, this
 * vdev is considered partially resilvered and should leave its DTL
 * entries intact. The comment in vdev_dtl_reassess() describes how we
 * excise the DTLs.
 */
static boolean_t
vdev_dtl_should_excise(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;
        dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;

        ASSERT0(scn->scn_phys.scn_errors);
        ASSERT0(vd->vdev_children);

        if (vd->vdev_resilver_txg == 0 ||
            range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0)
                return (B_TRUE);

        /*
         * When a resilver is initiated the scan will assign the scn_max_txg
         * value to the highest txg value that exists in all DTLs. If this
         * device’s max DTL is not part of this scan (i.e. it is not in
         * the range (scn_min_txg, scn_max_txg] then it is not eligible
         * for excision.
         */
        if (vdev_dtl_max(vd) <= scn->scn_phys.scn_max_txg) {
                ASSERT3U(scn->scn_phys.scn_min_txg, <=, vdev_dtl_min(vd));
                ASSERT3U(scn->scn_phys.scn_min_txg, <, vd->vdev_resilver_txg);
                ASSERT3U(vd->vdev_resilver_txg, <=, scn->scn_phys.scn_max_txg);
                return (B_TRUE);
        }
        return (B_FALSE);
}

/*
 * Reassess DTLs after a config change or scrub completion.
 */
void
vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg, int scrub_done)
{
        spa_t *spa = vd->vdev_spa;
        avl_tree_t reftree;
        int minref;

        ASSERT(spa_config_held(spa, SCL_ALL, RW_READER) != 0);

        for (int c = 0; c < vd->vdev_children; c++)
                vdev_dtl_reassess(vd->vdev_child[c], txg,
                    scrub_txg, scrub_done);

        if (vd == spa->spa_root_vdev || vd->vdev_ishole || vd->vdev_aux)
                return;

        if (vd->vdev_ops->vdev_op_leaf) {
                dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;

                mutex_enter(&vd->vdev_dtl_lock);

                /*
                 * If we’ve completed a scan cleanly then determine
                 * if this vdev should remove any DTLs. We only want to
                 * excise regions on vdevs that were available during
                 * the entire duration of this scan.
                 */
                if (scrub_txg != 0 &&
                    (spa->spa_scrub_started ||
                    (scn != NULL && scn->scn_phys.scn_errors == 0)) &&
                    vdev_dtl_should_excise(vd)) {
                        /*
                         * We completed a scrub up to scrub_txg.  If we
                         * did it without rebooting, then the scrub dtl
                         * will be valid, so excise the old region and
                         * fold in the scrub dtl.  Otherwise, leave the
                         * dtl as-is if there was an error.
                         *
                         * There’s little trick here: to excise the beginning
                         * of the DTL_MISSING map, we put it into a reference
                         * tree and then add a segment with refcnt -1 that
```

```
                         * covers the range [0, scrub_txg).  This means
                         * that each txg in that range has refcnt -1 or 0.
                         * We then add DTL_SCRUB with a refcnt of 2, so that
                         * entries in the range [0, scrub_txg) will have a
                         * positive refcnt -- either 1 or 2.  We then convert
                         * the reference tree into the new DTL_MISSING map.
                         */
                        space_reftree_create(&reftree);
                        space_reftree_add_map(&reftree,
                            vd->vdev_dtl[DTL_MISSING], 1);
                        space_reftree_add_seg(&reftree, 0, scrub_txg, -1);
                        space_reftree_add_map(&reftree,
                            vd->vdev_dtl[DTL_SCRUB], 2);
                        space_reftree_generate_map(&reftree,
                            vd->vdev_dtl[DTL_MISSING], 1);
                        space_reftree_destroy(&reftree);
                }
                range_tree_vacate(vd->vdev_dtl[DTL_PARTIAL], NULL, NULL);
                range_tree_walk(vd->vdev_dtl[DTL_MISSING],
                    range_tree_add, vd->vdev_dtl[DTL_PARTIAL]);
                if (scrub_done)
                        range_tree_vacate(vd->vdev_dtl[DTL_SCRUB], NULL, NULL);
                range_tree_vacate(vd->vdev_dtl[DTL_OUTAGE], NULL, NULL);
                if (!vdev_readable(vd))
                        range_tree_add(vd->vdev_dtl[DTL_OUTAGE], 0, -1ULL);
                else
                        range_tree_walk(vd->vdev_dtl[DTL_MISSING],
                            range_tree_add, vd->vdev_dtl[DTL_OUTAGE]);

                /*
                 * If the vdev was resilvering and no longer has any
                 * DTLs then reset its resilvering flag.
                 */
                if (vd->vdev_resilver_txg != 0 &&
                    range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0 &&
                    range_tree_space(vd->vdev_dtl[DTL_OUTAGE]) == 0)
                        vd->vdev_resilver_txg = 0;

                mutex_exit(&vd->vdev_dtl_lock);

                if (txg != 0)
                        vdev_dirty(vd->vdev_top, VDD_DTL, vd, txg);
                return;
        }

        mutex_enter(&vd->vdev_dtl_lock);
        for (int t = 0; t < DTL_TYPES; t++) {
                /* account for child’s outage in parent’s missing map */
                int s = (t == DTL_MISSING) ? DTL_OUTAGE: t;
                if (t == DTL_SCRUB)
                        continue;                       /* leaf vdevs only */
                if (t == DTL_PARTIAL)
                        minref = 1;                     /* i.e. non-zero */
                else if (vd->vdev_nparity != 0)
                        minref = vd->vdev_nparity + 1;  /* RAID-Z */
                else
                        minref = vd->vdev_children;     /* any kind of mirror */
                space_reftree_create(&reftree);
                for (int c = 0; c < vd->vdev_children; c++) {
                        vdev_t *cvd = vd->vdev_child[c];
                        mutex_enter(&cvd->vdev_dtl_lock);
                        space_reftree_add_map(&reftree, cvd->vdev_dtl[s], 1);
                        mutex_exit(&cvd->vdev_dtl_lock);
                }
                space_reftree_generate_map(&reftree, vd->vdev_dtl[t], minref);
                space_reftree_destroy(&reftree);
        }
        mutex_exit(&vd->vdev_dtl_lock);
}

int
vdev_dtl_load(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;
        objset_t *mos = spa->spa_meta_objset;
        int error = 0;

        if (vd->vdev_ops->vdev_op_leaf && vd->vdev_dtl_object != 0) {
                ASSERT(!vd->vdev_ishole);

                error = space_map_open(&vd->vdev_dtl_sm, mos,
                    vd->vdev_dtl_object, 0, -1ULL, 0, &vd->vdev_dtl_lock);
                if (error)
                        return (error);
                ASSERT(vd->vdev_dtl_sm != NULL);

                mutex_enter(&vd->vdev_dtl_lock);

                /*
                 * Now that we’ve opened the space_map we need to update
                 * the in-core DTL.
                 */
                space_map_update(vd->vdev_dtl_sm);

                error = space_map_load(vd->vdev_dtl_sm,
                    vd->vdev_dtl[DTL_MISSING], SM_ALLOC);
                mutex_exit(&vd->vdev_dtl_lock);

                return (error);
        }

        for (int c = 0; c < vd->vdev_children; c++) {
                error = vdev_dtl_load(vd->vdev_child[c]);
                if (error != 0)
                        break;
        }

        return (error);
}

void
vdev_dtl_sync(vdev_t *vd, uint64_t txg)
{
        spa_t *spa = vd->vdev_spa;
        range_tree_t *rt = vd->vdev_dtl[DTL_MISSING];
        objset_t *mos = spa->spa_meta_objset;
        range_tree_t *rtsync;
        kmutex_t rtlock;
        dmu_tx_t *tx;
        uint64_t object = space_map_object(vd->vdev_dtl_sm);

        ASSERT(!vd->vdev_ishole);
        ASSERT(vd->vdev_ops->vdev_op_leaf);

        tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);

        if (vd->vdev_detached || vd->vdev_top->vdev_removing) {
                mutex_enter(&vd->vdev_dtl_lock);
                space_map_free(vd->vdev_dtl_sm, tx);
                space_map_close(vd->vdev_dtl_sm);
                vd->vdev_dtl_sm = NULL;
                mutex_exit(&vd->vdev_dtl_lock);
```

```
                dmu_tx_commit(tx);
                return;
        }

        if (vd->vdev_dtl_sm == NULL) {
                uint64_t new_object;

                new_object = space_map_alloc(mos, tx);
                VERIFY3U(new_object, !=, 0);

                VERIFY0(space_map_open(&vd->vdev_dtl_sm, mos, new_object,
                    0, -1ULL, 0, &vd->vdev_dtl_lock));
                ASSERT(vd->vdev_dtl_sm != NULL);
        }

        mutex_init(&rtlock, NULL, MUTEX_DEFAULT, NULL);

        rtsync = range_tree_create(NULL, NULL, &rtlock);

        mutex_enter(&rtlock);

        mutex_enter(&vd->vdev_dtl_lock);
        range_tree_walk(rt, range_tree_add, rtsync);
        mutex_exit(&vd->vdev_dtl_lock);

        space_map_truncate(vd->vdev_dtl_sm, tx);
        space_map_write(vd->vdev_dtl_sm, rtsync, SM_ALLOC, tx);
        range_tree_vacate(rtsync, NULL, NULL);

        range_tree_destroy(rtsync);

        mutex_exit(&rtlock);
        mutex_destroy(&rtlock);

        /*
         * If the object for the space map has changed then dirty
         * the top level so that we update the config.
         */
        if (object != space_map_object(vd->vdev_dtl_sm)) {
                zfs_dbgmsg("txg %llu, spa %s, DTL old object %llu, "
                    "new object %llu", txg, spa_name(spa), object,
                    space_map_object(vd->vdev_dtl_sm));
                vdev_config_dirty(vd->vdev_top);
        }

        dmu_tx_commit(tx);

        mutex_enter(&vd->vdev_dtl_lock);
        space_map_update(vd->vdev_dtl_sm);
        mutex_exit(&vd->vdev_dtl_lock);
}

/*
 * Determine whether the specified vdev can be offlined/detached/removed
 * without losing data.
 */
boolean_t
vdev_dtl_required(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;
        vdev_t *tvd = vd->vdev_top;
        uint8_t cant_read = vd->vdev_cant_read;
        boolean_t required;

        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
        if (vd == spa->spa_root_vdev || vd == tvd)
                return (B_TRUE);

        /*
         * Temporarily mark the device as unreadable, and then determine
         * whether this results in any DTL outages in the top-level vdev.
         * If not, we can safely offline/detach/remove the device.
         */
        vd->vdev_cant_read = B_TRUE;
        vdev_dtl_reassess(tvd, 0, 0, B_FALSE);
        required = !vdev_dtl_empty(tvd, DTL_OUTAGE);
        vd->vdev_cant_read = cant_read;
        vdev_dtl_reassess(tvd, 0, 0, B_FALSE);

        if (!required && zio_injection_enabled)
                required = !!zio_handle_device_injection(vd, NULL, ECHILD);

        return (required);
}

/*
 * Determine if resilver is needed, and if so the txg range.
 */
boolean_t
vdev_resilver_needed(vdev_t *vd, uint64_t *minp, uint64_t *maxp)
{
        boolean_t needed = B_FALSE;
        uint64_t thismin = UINT64_MAX;
        uint64_t thismax = 0;

        if (vd->vdev_children == 0) {
                mutex_enter(&vd->vdev_dtl_lock);
                if (range_tree_space(vd->vdev_dtl[DTL_MISSING]) != 0 &&
                    vdev_writeable(vd)) {

                        thismin = vdev_dtl_min(vd);
                        thismax = vdev_dtl_max(vd);
                        needed = B_TRUE;
                }
                mutex_exit(&vd->vdev_dtl_lock);
        } else {
                for (int c = 0; c < vd->vdev_children; c++) {
                        vdev_t *cvd = vd->vdev_child[c];
                        uint64_t cmin, cmax;

                        if (vdev_resilver_needed(cvd, &cmin, &cmax)) {
                                thismin = MIN(thismin, cmin);
                                thismax = MAX(thismax, cmax);
                                needed = B_TRUE;
                        }
                }
        }

        if (needed && minp) {
                *minp = thismin;
                *maxp = thismax;
        }
        return (needed);
}

void
vdev_load(vdev_t *vd)
{
        /*
         * Recursively load all children.
         */
```

```
        for (int c = 0; c < vd->vdev_children; c++)
                vdev_load(vd->vdev_child[c]);

        /*
         * If this is a top-level vdev, initialize its metaslabs.
         */
        if (vd == vd->vdev_top && !vd->vdev_ishole &&
            (vd->vdev_ashift == 0 || vd->vdev_asize == 0 ||
            vdev_metaslab_init(vd, 0) != 0))
                vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
                    VDEV_AUX_CORRUPT_DATA);

        /*
         * If this is a leaf vdev, load its DTL.
         */
        if (vd->vdev_ops->vdev_op_leaf && vdev_dtl_load(vd) != 0)
                vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
                    VDEV_AUX_CORRUPT_DATA);
}

/*
 * The special vdev case is used for hot spares and l2cache devices.  Its
 * sole purpose it to set the vdev state for the associated vdev.  To do this,
 * we make sure that we can open the underlying device, then try to read the
 * label, and make sure that the label is sane and that it hasn’t been
 * repurposed to another pool.
 */
int
vdev_validate_aux(vdev_t *vd)
{
        nvlist_t *label;
        uint64_t guid, version;
        uint64_t state;

        if (!vdev_readable(vd))
                return (0);

        if ((label = vdev_label_read_config(vd, -1ULL)) == NULL) {
                vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
                    VDEV_AUX_CORRUPT_DATA);
                return (-1);
        }

        if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_VERSION, &version) != 0 ||
            !SPA_VERSION_IS_SUPPORTED(version) ||
            nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID, &guid) != 0 ||
            guid != vd->vdev_guid ||
            nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE, &state) != 0) {
                vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
                    VDEV_AUX_CORRUPT_DATA);
                nvlist_free(label);
                return (-1);
        }

        /*
         * We don’t actually check the pool state here.  If it’s in fact in
         * use by another pool, we update this fact on the fly when requested.
         */
        nvlist_free(label);
        return (0);
}

void
vdev_remove(vdev_t *vd, uint64_t txg)
{
        spa_t *spa = vd->vdev_spa;
        objset_t *mos = spa->spa_meta_objset;
        dmu_tx_t *tx;

        tx = dmu_tx_create_assigned(spa_get_dsl(spa), txg);

        if (vd->vdev_ms != NULL) {
                metaslab_group_t *mg = vd->vdev_mg;

                metaslab_group_histogram_verify(mg);
                metaslab_class_histogram_verify(mg->mg_class);

                for (int m = 0; m < vd->vdev_ms_count; m++) {
                        metaslab_t *msp = vd->vdev_ms[m];

                        if (msp == NULL || msp->ms_sm == NULL)
                                continue;

                        mutex_enter(&msp->ms_lock);
                        /*
                         * If the metaslab was not loaded when the vdev
                         * was removed then the histogram accounting may
                         * not be accurate. Update the histogram information
                         * here so that we ensure that the metaslab group
                         * and metaslab class are up-to-date.
                         */
                        metaslab_group_histogram_remove(mg, msp);

                        VERIFY0(space_map_allocated(msp->ms_sm));
                        space_map_free(msp->ms_sm, tx);
                        space_map_close(msp->ms_sm);
                        msp->ms_sm = NULL;
                        mutex_exit(&msp->ms_lock);
                }

                metaslab_group_histogram_verify(mg);
                metaslab_class_histogram_verify(mg->mg_class);
                for (int i = 0; i < RANGE_TREE_HISTOGRAM_SIZE; i++)
                        ASSERT0(mg->mg_histogram[i]);

        }

        if (vd->vdev_ms_array) {
                (void) dmu_object_free(mos, vd->vdev_ms_array, tx);
                vd->vdev_ms_array = 0;
        }
        dmu_tx_commit(tx);
}

void
vdev_sync_done(vdev_t *vd, uint64_t txg)
{
        metaslab_t *msp;
        boolean_t reassess = !txg_list_empty(&vd->vdev_ms_list, TXG_CLEAN(txg));

        ASSERT(!vd->vdev_ishole);

        while (msp = txg_list_remove(&vd->vdev_ms_list, TXG_CLEAN(txg)))
                metaslab_sync_done(msp, txg);

        if (reassess)
                metaslab_sync_reassess(vd->vdev_mg);
}

void
vdev_sync(vdev_t *vd, uint64_t txg)
{
```

```
        spa_t *spa = vd->vdev_spa;
        vdev_t *lvd;
        metaslab_t *msp;
        dmu_tx_t *tx;

        ASSERT(!vd->vdev_ishole);

        if (vd->vdev_ms_array == 0 && vd->vdev_ms_shift != 0) {
                ASSERT(vd == vd->vdev_top);
                tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);
                vd->vdev_ms_array = dmu_object_alloc(spa->spa_meta_objset,
                    DMU_OT_OBJECT_ARRAY, 0, DMU_OT_NONE, 0, tx);
                ASSERT(vd->vdev_ms_array != 0);
                vdev_config_dirty(vd);
                dmu_tx_commit(tx);
        }

        /*
         * Remove the metadata associated with this vdev once it's empty.
         */
        if (vd->vdev_stat.vs_alloc == 0 && vd->vdev_removing)
                vdev_remove(vd, txg);

        while ((msp = txg_list_remove(&vd->vdev_ms_list, txg)) != NULL) {
                metaslab_sync(msp, txg);
                (void) txg_list_add(&vd->vdev_ms_list, msp, TXG_CLEAN(txg));
        }

        while ((lvd = txg_list_remove(&vd->vdev_dtl_list, txg)) != NULL)
                vdev_dtl_sync(lvd, txg);

        (void) txg_list_add(&spa->spa_vdev_txg_list, vd, TXG_CLEAN(txg));
}

uint64_t
vdev_psize_to_asize(vdev_t *vd, uint64_t psize)
{
        return (vd->vdev_ops->vdev_op_asize(vd, psize));
}

/*
 * Mark the given vdev faulted.  A faulted vdev behaves as if the device could
 * not be opened, and no I/O is attempted.
 */
int
vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux)
{
        vdev_t *vd, *tvd;

        spa_vdev_state_enter(spa, SCL_NONE);

        if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
                return (spa_vdev_state_exit(spa, NULL, ENODEV));

        if (!vd->vdev_ops->vdev_op_leaf)
                return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

        tvd = vd->vdev_top;

        /*
         * We don't directly use the aux state here, but if we do a
         * vdev_reopen(), we need this value to be present to remember why we
         * were faulted.
         */
        vd->vdev_label_aux = aux;
        /*
         * Faulted state takes precedence over degraded.
         */
        vd->vdev_delayed_close = B_FALSE;
        vd->vdev_faulted = 1ULL;
        vd->vdev_degraded = 0ULL;
        vdev_set_state(vd, B_FALSE, VDEV_STATE_FAULTED, aux);

        /*
         * If this device has the only valid copy of the data, then
         * back off and simply mark the vdev as degraded instead.
         */
        if (!tvd->vdev_islog && vd->vdev_aux == NULL && vdev_dtl_required(vd)) {
                vd->vdev_degraded = 1ULL;
                vd->vdev_faulted = 0ULL;

                /*
                 * If we reopen the device and it's not dead, only then do we
                 * mark it degraded.
                 */
                vdev_reopen(tvd);

                if (vdev_readable(vd))
                        vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED, aux);
        }

        return (spa_vdev_state_exit(spa, vd, 0));
}

/*
 * Mark the given vdev degraded.  A degraded vdev is purely an indication to the
 * user that something is wrong.  The vdev continues to operate as normal as far
 * as I/O is concerned.
 */
int
vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux)
{
        vdev_t *vd;

        spa_vdev_state_enter(spa, SCL_NONE);

        if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
                return (spa_vdev_state_exit(spa, NULL, ENODEV));

        if (!vd->vdev_ops->vdev_op_leaf)
                return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

        /*
         * If the vdev is already faulted, then don't do anything.
         */
        if (vd->vdev_faulted || vd->vdev_degraded)
                return (spa_vdev_state_exit(spa, NULL, 0));

        vd->vdev_degraded = 1ULL;
        if (!vdev_is_dead(vd))
                vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED,
                    aux);

        return (spa_vdev_state_exit(spa, vd, 0));
}

/*
 * Online the given vdev.
 *
 * If 'ZFS_ONLINE_UNSPARE' is set, it implies two things.  First, any attached
 * spare device should be detached when the device finishes resilvering.
```

```
 * Second, the online should be treated like a 'test' online case, so no FMA
 * events are generated if the device fails to open.
 */
int
vdev_online(spa_t *spa, uint64_t guid, uint64_t flags, vdev_state_t *newstate)
{
        vdev_t *vd, *tvd, *pvd, *rvd = spa->spa_root_vdev;

        spa_vdev_state_enter(spa, SCL_NONE);

        if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
                return (spa_vdev_state_exit(spa, NULL, ENODEV));

        if (!vd->vdev_ops->vdev_op_leaf)
                return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

        tvd = vd->vdev_top;
        vd->vdev_offline = B_FALSE;
        vd->vdev_tmpoffline = B_FALSE;
        vd->vdev_checkremove = !!(flags & ZFS_ONLINE_CHECKREMOVE);
        vd->vdev_forcefault = !!(flags & ZFS_ONLINE_FORCEFAULT);

        /* XXX - L2ARC 1.0 does not support expansion */
        if (!vd->vdev_aux) {
                for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
                        pvd->vdev_expanding = !!(flags & ZFS_ONLINE_EXPAND);
        }

        vdev_reopen(tvd);
        vd->vdev_checkremove = vd->vdev_forcefault = B_FALSE;

        if (!vd->vdev_aux) {
                for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
                        pvd->vdev_expanding = B_FALSE;
        }

        if (newstate)
                *newstate = vd->vdev_state;
        if ((flags & ZFS_ONLINE_UNSPARE) &&
            !vdev_is_dead(vd) && vd->vdev_parent &&
            vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
            vd->vdev_parent->vdev_child[0] == vd)
                vd->vdev_unspare = B_TRUE;

        if ((flags & ZFS_ONLINE_EXPAND) || spa->spa_autoexpand) {

                /* XXX - L2ARC 1.0 does not support expansion */
                if (vd->vdev_aux)
                        return (spa_vdev_state_exit(spa, vd, ENOTSUP));
                spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
        }
        return (spa_vdev_state_exit(spa, vd, 0));
}

static int
vdev_offline_locked(spa_t *spa, uint64_t guid, uint64_t flags)
{
        vdev_t *vd, *tvd;
        int error = 0;
        uint64_t generation;
        metaslab_group_t *mg;

top:
        spa_vdev_state_enter(spa, SCL_ALLOC);

        if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
```

```
                return (spa_vdev_state_exit(spa, NULL, ENODEV));

        if (!vd->vdev_ops->vdev_op_leaf)
                return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

        tvd = vd->vdev_top;
        mg = tvd->vdev_mg;
        generation = spa->spa_config_generation + 1;

        /*
         * If the device isn’t already offline, try to offline it.
         */
        if (!vd->vdev_offline) {
                /*
                 * If this device has the only valid copy of some data,
                 * don’t allow it to be offlined. Log devices are always
                 * expendable.
                 */
                if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
                    vdev_dtl_required(vd))
                        return (spa_vdev_state_exit(spa, NULL, EBUSY));

                /*
                 * If the top-level is a slog and it has had allocations
                 * then proceed.  We check that the vdev’s metaslab group
                 * is not NULL since it’s possible that we may have just
                 * added this vdev but not yet initialized its metaslabs.
                 */
                if (tvd->vdev_islog && mg != NULL) {
                        /*
                         * Prevent any future allocations.
                         */
                        metaslab_group_passivate(mg);
                        (void) spa_vdev_state_exit(spa, vd, 0);

                        error = spa_offline_log(spa);

                        spa_vdev_state_enter(spa, SCL_ALLOC);

                        /*
                         * Check to see if the config has changed.
                         */
                        if (error || generation != spa->spa_config_generation) {
                                metaslab_group_activate(mg);
                                if (error)
                                        return (spa_vdev_state_exit(spa,
                                            vd, error));
                                (void) spa_vdev_state_exit(spa, vd, 0);
                                goto top;
                        }
                        ASSERT0(tvd->vdev_stat.vs_alloc);
                }

                /*
                 * Offline this device and reopen its top-level vdev.
                 * If the top-level vdev is a log device then just offline
                 * it. Otherwise, if this action results in the top-level
                 * vdev becoming unusable, undo it and fail the request.
                 */
                vd->vdev_offline = B_TRUE;
                vdev_reopen(tvd);

                if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
                    vdev_is_dead(tvd)) {
                        vd->vdev_offline = B_FALSE;
                        vdev_reopen(tvd);
```

```
                        return (spa_vdev_state_exit(spa, NULL, EBUSY));
                }

                /*
                 * Add the device back into the metaslab rotor so that
                 * once we online the device it’s open for business.
                 */
                if (tvd->vdev_islog && mg != NULL)
                        metaslab_group_activate(mg);
        }

        vd->vdev_tmpoffline = !!(flags & ZFS_OFFLINE_TEMPORARY);

        return (spa_vdev_state_exit(spa, vd, 0));
}

int
vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags)
{
        int error;

        mutex_enter(&spa->spa_vdev_top_lock);
        error = vdev_offline_locked(spa, guid, flags);
        mutex_exit(&spa->spa_vdev_top_lock);

        return (error);
}

/*
 * Clear the error counts associated with this vdev.  Unlike vdev_online() and
 * vdev_offline(), we assume the spa config is locked.  We also clear all
 * children.  If ’vd’ is NULL, then the user wants to clear all vdevs.
 */
void
vdev_clear(spa_t *spa, vdev_t *vd)
{
        vdev_t *rvd = spa->spa_root_vdev;

        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

        if (vd == NULL)
                vd = rvd;

        vd->vdev_stat.vs_read_errors = 0;
        vd->vdev_stat.vs_write_errors = 0;
        vd->vdev_stat.vs_checksum_errors = 0;

        for (int c = 0; c < vd->vdev_children; c++)
                vdev_clear(spa, vd->vdev_child[c]);

        /*
         * If we’re in the FAULTED state or have experienced failed I/O, then
         * clear the persistent state and attempt to reopen the device.  We
         * also mark the vdev config dirty, so that the new faulted state is
         * written out to disk.
         */
        if (vd->vdev_faulted || vd->vdev_degraded ||
            !vdev_readable(vd) || !vdev_writeable(vd)) {

                /*
                 * When reopening in reponse to a clear event, it may be due to
                 * a fmadm repair request.  In this case, if the device is
                 * still broken, we want to still post the ereport again.
                 */
                vd->vdev_forcefault = B_TRUE;
                vd->vdev_faulted = vd->vdev_degraded = 0ULL;
                vd->vdev_cant_read = B_FALSE;
                vd->vdev_cant_write = B_FALSE;

                vdev_reopen(vd == rvd ? rvd : vd->vdev_top);

                vd->vdev_forcefault = B_FALSE;

                if (vd != rvd && vdev_writeable(vd->vdev_top))
                        vdev_state_dirty(vd->vdev_top);

                if (vd->vdev_aux == NULL && !vdev_is_dead(vd))
                        spa_async_request(spa, SPA_ASYNC_RESILVER);

                spa_event_notify(spa, vd, ESC_ZFS_VDEV_CLEAR);
        }

        /*
         * When clearing a FMA-diagnosed fault, we always want to
         * unspare the device, as we assume that the original spare was
         * done in response to the FMA fault.
         */
        if (!vdev_is_dead(vd) && vd->vdev_parent != NULL &&
            vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
            vd->vdev_parent->vdev_child[0] == vd)
                vd->vdev_unspare = B_TRUE;
}

boolean_t
vdev_is_dead(vdev_t *vd)
{
        /*
         * Holes and missing devices are always considered "dead".
         * This simplifies the code since we don’t have to check for
         * these types of devices in the various code paths.
         * Instead we rely on the fact that we skip over dead devices
         * before issuing I/O to them.
         */
        return (vd->vdev_state < VDEV_STATE_DEGRADED || vd->vdev_ishole ||
            vd->vdev_ops == &vdev_missing_ops);
}

boolean_t
vdev_readable(vdev_t *vd)
{
        return (!vdev_is_dead(vd) && !vd->vdev_cant_read);
}

boolean_t
vdev_writeable(vdev_t *vd)
{
        return (!vdev_is_dead(vd) && !vd->vdev_cant_write);
}

boolean_t
vdev_allocatable(vdev_t *vd)
{
        uint64_t state = vd->vdev_state;

        /*
         * We currently allow allocations from vdevs which may be in the
         * process of reopening (i.e. VDEV_STATE_CLOSED). If the device
         * fails to reopen then we’ll catch it later when we’re holding
         * the proper locks.  Note that we have to get the vdev state
         * in a local variable because although it changes atomically,
         * we’re asking two separate questions about it.
```

```
         */
        return (!(state < VDEV_STATE_DEGRADED && state != VDEV_STATE_CLOSED) &&
            !vd->vdev_cant_write && !vd->vdev_ishole);
}

boolean_t
vdev_accessible(vdev_t *vd, zio_t *zio)
{
        ASSERT(zio->io_vd == vd);

        if (vdev_is_dead(vd) || vd->vdev_remove_wanted)
                return (B_FALSE);

        if (zio->io_type == ZIO_TYPE_READ)
                return (!vd->vdev_cant_read);

        if (zio->io_type == ZIO_TYPE_WRITE)
                return (!vd->vdev_cant_write);

        return (B_TRUE);
}

/*
 * Get statistics for the given vdev.
 */
void
vdev_get_stats(vdev_t *vd, vdev_stat_t *vs)
{
        spa_t *spa = vd->vdev_spa;
        vdev_t *rvd = spa->spa_root_vdev;

        ASSERT(spa_config_held(spa, SCL_ALL, RW_READER) != 0);

        mutex_enter(&vd->vdev_stat_lock);
        bcopy(&vd->vdev_stat, vs, sizeof (*vs));
        vs->vs_timestamp = gethrtime() - vs->vs_timestamp;
        vs->vs_state = vd->vdev_state;
        vs->vs_rsize = vdev_get_min_asize(vd);
        if (vd->vdev_ops->vdev_op_leaf)
                vs->vs_rsize += VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE;
        vs->vs_esize = vd->vdev_max_asize - vd->vdev_asize;
        if (vd->vdev_aux == NULL && vd == vd->vdev_top && !vd->vdev_ishole) {
                vs->vs_fragmentation = vd->vdev_mg->mg_fragmentation;
        }

        /*
         * If we’re getting stats on the root vdev, aggregate the I/O counts
         * over all top-level vdevs (i.e. the direct children of the root).
         */
        if (vd == rvd) {
                for (int c = 0; c < rvd->vdev_children; c++) {
                        vdev_t *cvd = rvd->vdev_child[c];
                        vdev_stat_t *cvs = &cvd->vdev_stat;

                        for (int t = 0; t < ZIO_TYPES; t++) {
                                vs->vs_ops[t] += cvs->vs_ops[t];
                                vs->vs_bytes[t] += cvs->vs_bytes[t];
                        }
                        cvs->vs_scan_removing = cvd->vdev_removing;
                }
        }
        mutex_exit(&vd->vdev_stat_lock);
}

void
vdev_clear_stats(vdev_t *vd)
```

```
{
        mutex_enter(&vd->vdev_stat_lock);
        vd->vdev_stat.vs_space = 0;
        vd->vdev_stat.vs_dspace = 0;
        vd->vdev_stat.vs_alloc = 0;
        mutex_exit(&vd->vdev_stat_lock);
}

void
vdev_scan_stat_init(vdev_t *vd)
{
        vdev_stat_t *vs = &vd->vdev_stat;

        for (int c = 0; c < vd->vdev_children; c++)
                vdev_scan_stat_init(vd->vdev_child[c]);

        mutex_enter(&vd->vdev_stat_lock);
        vs->vs_scan_processed = 0;
        mutex_exit(&vd->vdev_stat_lock);
}

void
vdev_stat_update(zio_t *zio, uint64_t psize)
{
        spa_t *spa = zio->io_spa;
        vdev_t *rvd = spa->spa_root_vdev;
        vdev_t *vd = zio->io_vd ? zio->io_vd : rvd;
        vdev_t *pvd;
        uint64_t txg = zio->io_txg;
        vdev_stat_t *vs = &vd->vdev_stat;
        zio_type_t type = zio->io_type;
        int flags = zio->io_flags;

        /*
         * If this i/o is a gang leader, it didn’t do any actual work.
         */
        if (zio->io_gang_tree)
                return;

        if (zio->io_error == 0) {
                /*
                 * If this is a root i/o, don’t count it -- we’ve already
                 * counted the top-level vdevs, and vdev_get_stats() will
                 * aggregate them when asked.  This reduces contention on
                 * the root vdev_stat_lock and implicitly handles blocks
                 * that compress away to holes, for which there is no i/o.
                 * (Holes never create vdev children, so all the counters
                 * remain zero, which is what we want.)
                 *
                 * Note: this only applies to successful i/o (io_error == 0)
                 * because unlike i/o counts, errors are not additive.
                 * When reading a ditto block, for example, failure of
                 * one top-level vdev does not imply a root-level error.
                 */
                if (vd == rvd)
                        return;

                ASSERT(vd == zio->io_vd);

                if (flags & ZIO_FLAG_IO_BYPASS)
                        return;

                mutex_enter(&vd->vdev_stat_lock);

                if (flags & ZIO_FLAG_IO_REPAIR) {
                        if (flags & ZIO_FLAG_SCAN_THREAD) {
```

```
                                dsl_scan_phys_t *scn_phys =
                                    &spa->spa_dsl_pool->dp_scan->scn_phys;
                                uint64_t *processed = &scn_phys->scn_processed;

                                /* XXX cleanup? */
                                if (vd->vdev_ops->vdev_op_leaf)
                                        atomic_add_64(processed, psize);
                                vs->vs_scan_processed += psize;
                        }

                        if (flags & ZIO_FLAG_SELF_HEAL)
                                vs->vs_self_healed += psize;
                }

                vs->vs_ops[type]++;
                vs->vs_bytes[type] += psize;

                mutex_exit(&vd->vdev_stat_lock);
                return;
        }

        if (flags & ZIO_FLAG_SPECULATIVE)
                return;

        /*
         * If this is an I/O error that is going to be retried, then ignore the
         * error.  Otherwise, the user may interpret B_FAILFAST I/O errors as
         * hard errors, when in reality they can happen for any number of
         * innocuous reasons (bus resets, MPxIO link failure, etc).
         */
        if (zio->io_error == EIO &&
            !(zio->io_flags & ZIO_FLAG_IO_RETRY))
                return;

        /*
         * Intent logs writes won’t propagate their error to the root
         * I/O so don’t mark these types of failures as pool-level
         * errors.
         */
        if (zio->io_vd == NULL && (zio->io_flags & ZIO_FLAG_DONT_PROPAGATE))
                return;

        mutex_enter(&vd->vdev_stat_lock);
        if (type == ZIO_TYPE_READ && !vdev_is_dead(vd)) {
                if (zio->io_error == ECKSUM)
                        vs->vs_checksum_errors++;
                else
                        vs->vs_read_errors++;
        }
        if (type == ZIO_TYPE_WRITE && !vdev_is_dead(vd))
                vs->vs_write_errors++;
        mutex_exit(&vd->vdev_stat_lock);

        if (type == ZIO_TYPE_WRITE && txg != 0 &&
            (!(flags & ZIO_FLAG_IO_REPAIR) ||
            (flags & ZIO_FLAG_SCAN_THREAD) ||
            spa->spa_claiming)) {
                /*
                 * This is either a normal write (not a repair), or it’s
                 * a repair induced by the scrub thread, or it’s a repair
                 * made by zil_claim() during spa_load() in the first txg.
                 * In the normal case, we commit the DTL change in the same
                 * txg as the block was born.  In the scrub-induced repair
                 * case, we know that scrubs run in first-pass syncing context,
                 * so we commit the DTL change in spa_syncing_txg(spa).
                 * In the zil_claim() case, we commit in spa_first_txg(spa).
                 *
                 * We currently do not make DTL entries for failed spontaneous
                 * self-healing writes triggered by normal (non-scrubbing)
                 * reads, because we have no transactional context in which to
                 * do so -- and it's not clear that it'd be desirable anyway.
                 */
                if (vd->vdev_ops->vdev_op_leaf) {
                        uint64_t commit_txg = txg;
                        if (flags & ZIO_FLAG_SCAN_THREAD) {
                                ASSERT(flags & ZIO_FLAG_IO_REPAIR);
                                ASSERT(spa_sync_pass(spa) == 1);
                                vdev_dtl_dirty(vd, DTL_SCRUB, txg, 1);
                                commit_txg = spa_syncing_txg(spa);
                        } else if (spa->spa_claiming) {
                                ASSERT(flags & ZIO_FLAG_IO_REPAIR);
                                commit_txg = spa_first_txg(spa);
                        }
                        ASSERT(commit_txg >= spa_syncing_txg(spa));
                        if (vdev_dtl_contains(vd, DTL_MISSING, txg, 1))
                                return;
                        for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
                                vdev_dtl_dirty(pvd, DTL_PARTIAL, txg, 1);
                        vdev_dirty(vd->vdev_top, VDD_DTL, vd, commit_txg);
                }
                if (vd != rvd)
                        vdev_dtl_dirty(vd, DTL_MISSING, txg, 1);
        }
}

/*
 * Update the in-core space usage stats for this vdev, its metaslab class,
 * and the root vdev.
 */
void
vdev_space_update(vdev_t *vd, int64_t alloc_delta, int64_t defer_delta,
    int64_t space_delta)
{
        int64_t dspace_delta = space_delta;
        spa_t *spa = vd->vdev_spa;
        vdev_t *rvd = spa->spa_root_vdev;
        metaslab_group_t *mg = vd->vdev_mg;
        metaslab_class_t *mc = mg ? mg->mg_class : NULL;

        ASSERT(vd == vd->vdev_top);

        /*
         * Apply the inverse of the psize-to-asize (ie. RAID-Z) space-expansion
         * factor.  We must calculate this here and not at the root vdev
         * because the root vdev's psize-to-asize is simply the max of its
         * childrens', thus not accurate enough for us.
         */
        ASSERT((dspace_delta & (SPA_MINBLOCKSIZE-1)) == 0);
        ASSERT(vd->vdev_deflate_ratio != 0 || vd->vdev_isl2cache);
        dspace_delta = (dspace_delta >> SPA_MINBLOCKSHIFT) *
            vd->vdev_deflate_ratio;

        mutex_enter(&vd->vdev_stat_lock);
        vd->vdev_stat.vs_alloc += alloc_delta;
        vd->vdev_stat.vs_space += space_delta;
        vd->vdev_stat.vs_dspace += dspace_delta;
        mutex_exit(&vd->vdev_stat_lock);

        if (mc == spa_normal_class(spa)) {
                mutex_enter(&rvd->vdev_stat_lock);
                rvd->vdev_stat.vs_alloc += alloc_delta;
                rvd->vdev_stat.vs_space += space_delta;
```

```
                rvd->vdev_stat.vs_dspace += dspace_delta;
                mutex_exit(&rvd->vdev_stat_lock);
        }

        if (mc != NULL) {
                ASSERT(rvd == vd->vdev_parent);
                ASSERT(vd->vdev_ms_count != 0);

                metaslab_class_space_update(mc,
                    alloc_delta, defer_delta, space_delta, dspace_delta);
        }
}

/*
 * Mark a top-level vdev's config as dirty, placing it on the dirty list
 * so that it will be written out next time the vdev configuration is synced.
 * If the root vdev is specified (vdev_top == NULL), dirty all top-level vdevs.
 */
void
vdev_config_dirty(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;
        vdev_t *rvd = spa->spa_root_vdev;
        int c;

        ASSERT(spa_writeable(spa));

        /*
         * If this is an aux vdev (as with l2cache and spare devices), then we
         * update the vdev config manually and set the sync flag.
         */
        if (vd->vdev_aux != NULL) {
                spa_aux_vdev_t *sav = vd->vdev_aux;
                nvlist_t **aux;
                uint_t naux;

                for (c = 0; c < sav->sav_count; c++) {
                        if (sav->sav_vdevs[c] == vd)
                                break;
                }

                if (c == sav->sav_count) {
                        /*
                         * We're being removed.  There's nothing more to do.
                         */
                        ASSERT(sav->sav_sync == B_TRUE);
                        return;
                }

                sav->sav_sync = B_TRUE;

                if (nvlist_lookup_nvlist_array(sav->sav_config,
                    ZPOOL_CONFIG_L2CACHE, &aux, &naux) != 0) {
                        VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
                            ZPOOL_CONFIG_SPARES, &aux, &naux) == 0);
                }

                ASSERT(c < naux);

                /*
                 * Setting the nvlist in the middle if the array is a little
                 * sketchy, but it will work.
                 */
                nvlist_free(aux[c]);
                aux[c] = vdev_config_generate(spa, vd, B_TRUE, 0);
                return;
        }

        /*
         * The dirty list is protected by the SCL_CONFIG lock.  The caller
         * must either hold SCL_CONFIG as writer, or must be the sync thread
         * (which holds SCL_CONFIG as reader).  There’s only one sync thread,
         * so this is sufficient to ensure mutual exclusion.
         */
        ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
            (dsl_pool_sync_context(spa_get_dsl(spa)) &&
            spa_config_held(spa, SCL_CONFIG, RW_READER)));

        if (vd == rvd) {
                for (c = 0; c < rvd->vdev_children; c++)
                        vdev_config_dirty(rvd->vdev_child[c]);
        } else {
                ASSERT(vd == vd->vdev_top);

                if (!list_link_active(&vd->vdev_config_dirty_node) &&
                    !vd->vdev_ishole)
                        list_insert_head(&spa->spa_config_dirty_list, vd);
        }
}

void
vdev_config_clean(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;

        ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
            (dsl_pool_sync_context(spa_get_dsl(spa)) &&
            spa_config_held(spa, SCL_CONFIG, RW_READER)));

        ASSERT(list_link_active(&vd->vdev_config_dirty_node));
        list_remove(&spa->spa_config_dirty_list, vd);
}

/*
 * Mark a top-level vdev’s state as dirty, so that the next pass of
 * spa_sync() can convert this into vdev_config_dirty().  We distinguish
 * the state changes from larger config changes because they require
 * much less locking, and are often needed for administrative actions.
 */
void
vdev_state_dirty(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;

        ASSERT(spa_writeable(spa));
        ASSERT(vd == vd->vdev_top);

        /*
         * The state list is protected by the SCL_STATE lock.  The caller
         * must either hold SCL_STATE as writer, or must be the sync thread
         * (which holds SCL_STATE as reader).  There’s only one sync thread,
         * so this is sufficient to ensure mutual exclusion.
         */
        ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
            (dsl_pool_sync_context(spa_get_dsl(spa)) &&
            spa_config_held(spa, SCL_STATE, RW_READER)));

        if (!list_link_active(&vd->vdev_state_dirty_node) && !vd->vdev_ishole)
                list_insert_head(&spa->spa_state_dirty_list, vd);
}
```

```
void
vdev_state_clean(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;

        ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
            (dsl_pool_sync_context(spa_get_dsl(spa)) &&
            spa_config_held(spa, SCL_STATE, RW_READER)));

        ASSERT(list_link_active(&vd->vdev_state_dirty_node));
        list_remove(&spa->spa_state_dirty_list, vd);
}

/*
 * Propagate vdev state up from children to parent.
 */
void
vdev_propagate_state(vdev_t *vd)
{
        spa_t *spa = vd->vdev_spa;
        vdev_t *rvd = spa->spa_root_vdev;
        int degraded = 0, faulted = 0;
        int corrupted = 0;
        vdev_t *child;

        if (vd->vdev_children > 0) {
                for (int c = 0; c < vd->vdev_children; c++) {
                        child = vd->vdev_child[c];

                        /*
                         * Don’t factor holes into the decision.
                         */
                        if (child->vdev_ishole)
                                continue;

                        if (!vdev_readable(child) ||
                            (!vdev_writeable(child) && spa_writeable(spa))) {
                                /*
                                 * Root special: if there is a top-level log
                                 * device, treat the root vdev as if it were
                                 * degraded.
                                 */
                                if (child->vdev_islog && vd == rvd)
                                        degraded++;
                                else
                                        faulted++;
                        } else if (child->vdev_state <= VDEV_STATE_DEGRADED) {
                                degraded++;
                        }

                        if (child->vdev_stat.vs_aux == VDEV_AUX_CORRUPT_DATA)
                                corrupted++;
                }

                vd->vdev_ops->vdev_op_state_change(vd, faulted, degraded);

                /*
                 * Root special: if there is a top-level vdev that cannot be
                 * opened due to corrupted metadata, then propagate the root
                 * vdev’s aux state as ’corrupt’ rather than ’insufficient
                 * replicas’.
                 */
                if (corrupted && vd == rvd &&
                    rvd->vdev_state == VDEV_STATE_CANT_OPEN)
                        vdev_set_state(rvd, B_FALSE, VDEV_STATE_CANT_OPEN,
                            VDEV_AUX_CORRUPT_DATA);
        }

        if (vd->vdev_parent)
                vdev_propagate_state(vd->vdev_parent);
}

/*
 * Set a vdev's state.  If this is during an open, we don't update the parent
 * state, because we're in the process of opening children depth-first.
 * Otherwise, we propagate the change to the parent.
 *
 * If this routine places a device in a faulted state, an appropriate ereport is
 * generated.
 */
void
vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state, vdev_aux_t aux)
{
        uint64_t save_state;
        spa_t *spa = vd->vdev_spa;

        if (state == vd->vdev_state) {
                vd->vdev_stat.vs_aux = aux;
                return;
        }

        save_state = vd->vdev_state;

        vd->vdev_state = state;
        vd->vdev_stat.vs_aux = aux;

        /*
         * If we are setting the vdev state to anything but an open state, then
         * always close the underlying device unless the device has requested
         * a delayed close (i.e. we're about to remove or fault the device).
         * Otherwise, we keep accessible but invalid devices open forever.
         * We don't call vdev_close() itself, because that implies some extra
         * checks (offline, etc) that we don't want here.  This is limited to
         * leaf devices, because otherwise closing the device will affect other
         * children.
         */
        if (!vd->vdev_delayed_close && vdev_is_dead(vd) &&
            vd->vdev_ops->vdev_op_leaf)
                vd->vdev_ops->vdev_op_close(vd);

        /*
         * If we have brought this vdev back into service, we need
         * to notify fmd so that it can gracefully repair any outstanding
         * cases due to a missing device.  We do this in all cases, even those
         * that probably don't correlate to a repaired fault.  This is sure to
         * catch all cases, and we let the zfs-retire agent sort it out.  If
         * this is a transient state it's OK, as the retire agent will
         * double-check the state of the vdev before repairing it.
         */
        if (state == VDEV_STATE_HEALTHY && vd->vdev_ops->vdev_op_leaf &&
            vd->vdev_prevstate != state)
                zfs_post_state_change(spa, vd);

        if (vd->vdev_removed &&
            state == VDEV_STATE_CANT_OPEN &&
            (aux == VDEV_AUX_OPEN_FAILED || vd->vdev_checkremove)) {
                /*
                 * If the previous state is set to VDEV_STATE_REMOVED, then this
                 * device was previously marked removed and someone attempted to
                 * reopen it.  If this failed due to a nonexistent device, then
                 * keep the device in the REMOVED state.  We also let this be if
                 * it is one of our special test online cases, which is only
```

```
                 * attempting to online the device and shouldn't generate an FMA
                 * fault.
                 */
                vd->vdev_state = VDEV_STATE_REMOVED;
                vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
        } else if (state == VDEV_STATE_REMOVED) {
                vd->vdev_removed = B_TRUE;
        } else if (state == VDEV_STATE_CANT_OPEN) {
                /*
                 * If we fail to open a vdev during an import or recovery, we
                 * mark it as "not available", which signifies that it was
                 * never there to begin with.  Failure to open such a device
                 * is not considered an error.
                 */
                if ((spa_load_state(spa) == SPA_LOAD_IMPORT ||
                    spa_load_state(spa) == SPA_LOAD_RECOVER) &&
                    vd->vdev_ops->vdev_op_leaf)
                        vd->vdev_not_present = 1;

                /*
                 * Post the appropriate ereport.  If the 'prevstate' field is
                 * set to something other than VDEV_STATE_UNKNOWN, it indicates
                 * that this is part of a vdev_reopen().  In this case, we don't
                 * want to post the ereport if the device was already in the
                 * CANT_OPEN state beforehand.
                 *
                 * If the 'checkremove' flag is set, then this is an attempt to
                 * online the device in response to an insertion event.  If we
                 * hit this case, then we have detected an insertion event for a
                 * faulted or offline device that wasn't in the removed state.
                 * In this scenario, we don't post an ereport because we are
                 * about to replace the device, or attempt an online with
                 * vdev_forcefault, which will generate the fault for us.
                 */
                if ((vd->vdev_prevstate != state || vd->vdev_forcefault) &&
                    !vd->vdev_not_present && !vd->vdev_checkremove &&
                    vd != spa->spa_root_vdev) {
                        const char *class;

                        switch (aux) {
                        case VDEV_AUX_OPEN_FAILED:
                                class = FM_EREPORT_ZFS_DEVICE_OPEN_FAILED;
                                break;
                        case VDEV_AUX_CORRUPT_DATA:
                                class = FM_EREPORT_ZFS_DEVICE_CORRUPT_DATA;
                                break;
                        case VDEV_AUX_NO_REPLICAS:
                                class = FM_EREPORT_ZFS_DEVICE_NO_REPLICAS;
                                break;
                        case VDEV_AUX_BAD_GUID_SUM:
                                class = FM_EREPORT_ZFS_DEVICE_BAD_GUID_SUM;
                                break;
                        case VDEV_AUX_TOO_SMALL:
                                class = FM_EREPORT_ZFS_DEVICE_TOO_SMALL;
                                break;
                        case VDEV_AUX_BAD_LABEL:
                                class = FM_EREPORT_ZFS_DEVICE_BAD_LABEL;
                                break;
                        default:
                                class = FM_EREPORT_ZFS_DEVICE_UNKNOWN;
                        }

                        zfs_ereport_post(class, spa, vd, NULL, save_state, 0);
                }

                /* Erase any notion of persistent removed state */
                vd->vdev_removed = B_FALSE;
        } else {
                vd->vdev_removed = B_FALSE;
        }

        if (!isopen && vd->vdev_parent)
                vdev_propagate_state(vd->vdev_parent);
}

/*
 * Check the vdev configuration to ensure that it’s capable of supporting
 * a root pool. Currently, we do not support RAID-Z or partial configuration.
 * In addition, only a single top-level vdev is allowed and none of the leaves
 * can be wholedisks.
 */
boolean_t
vdev_is_bootable(vdev_t *vd)
{
        if (!vd->vdev_ops->vdev_op_leaf) {
                char *vdev_type = vd->vdev_ops->vdev_op_type;

                if (strcmp(vdev_type, VDEV_TYPE_ROOT) == 0 &&
                    vd->vdev_children > 1) {
                        return (B_FALSE);
                } else if (strcmp(vdev_type, VDEV_TYPE_RAIDZ) == 0 ||
                    strcmp(vdev_type, VDEV_TYPE_MISSING) == 0) {
                        return (B_FALSE);
                }
        }

        for (int c = 0; c < vd->vdev_children; c++) {
                if (!vdev_is_bootable(vd->vdev_child[c]))
                        return (B_FALSE);
        }
        return (B_TRUE);
}

/*
 * Load the state from the original vdev tree (ovd) which
 * we’ve retrieved from the MOS config object. If the original
 * vdev was offline or faulted then we transfer that state to the
 * device in the current vdev tree (nvd).
 */
void
vdev_load_log_state(vdev_t *nvd, vdev_t *ovd)
{
        spa_t *spa = nvd->vdev_spa;

        ASSERT(nvd->vdev_top->vdev_islog);
        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
        ASSERT3U(nvd->vdev_guid, ==, ovd->vdev_guid);

        for (int c = 0; c < nvd->vdev_children; c++)
                vdev_load_log_state(nvd->vdev_child[c], ovd->vdev_child[c]);

        if (nvd->vdev_ops->vdev_op_leaf) {
                /*
                 * Restore the persistent vdev state
                 */
                nvd->vdev_offline = ovd->vdev_offline;
                nvd->vdev_faulted = ovd->vdev_faulted;
                nvd->vdev_degraded = ovd->vdev_degraded;
                nvd->vdev_removed = ovd->vdev_removed;
        }
}
```

```
/*
 * Determine if a log device has valid content.  If the vdev was
 * removed or faulted in the MOS config then we know that
 * the content on the log device has already been written to the pool.
 */
boolean_t
vdev_log_state_valid(vdev_t *vd)
{
        if (vd->vdev_ops->vdev_op_leaf && !vd->vdev_faulted &&
            !vd->vdev_removed)
                return (B_TRUE);

        for (int c = 0; c < vd->vdev_children; c++)
                if (vdev_log_state_valid(vd->vdev_child[c]))
                        return (B_TRUE);

        return (B_FALSE);
}

/*
 * Expand a vdev if possible.
 */
void
vdev_expand(vdev_t *vd, uint64_t txg)
{
        ASSERT(vd->vdev_top == vd);
        ASSERT(spa_config_held(vd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

        if ((vd->vdev_asize >> vd->vdev_ms_shift) > vd->vdev_ms_count) {
                VERIFY(vdev_metaslab_init(vd, txg) == 0);
                vdev_config_dirty(vd);
        }
}

/*
 * Split a vdev.
 */
void
vdev_split(vdev_t *vd)
{
        vdev_t *cvd, *pvd = vd->vdev_parent;

        vdev_remove_child(pvd, vd);
        vdev_compact_children(pvd);

        cvd = pvd->vdev_child[0];
        if (pvd->vdev_children == 1) {
                vdev_remove_parent(cvd);
                cvd->vdev_splitting = B_TRUE;
        }
        vdev_propagate_state(cvd);
}

void
vdev_deadman(vdev_t *vd)
{
        for (int c = 0; c < vd->vdev_children; c++) {
                vdev_t *cvd = vd->vdev_child[c];

                vdev_deadman(cvd);
        }

        if (vd->vdev_ops->vdev_op_leaf) {
                vdev_queue_t *vq = &vd->vdev_queue;

                mutex_enter(&vq->vq_lock);
                if (avl_numnodes(&vq->vq_active_tree) > 0) {
                        spa_t *spa = vd->vdev_spa;
                        zio_t *fio;
                        uint64_t delta;

                        /*
                         * Look at the head of all the pending queues,
                         * if any I/O has been outstanding for longer than
                         * the spa_deadman_synctime we panic the system.
                         */
                        fio = avl_first(&vq->vq_active_tree);
                        delta = gethrtime() - fio->io_timestamp;
                        if (delta > spa_deadman_synctime(spa)) {
                                zfs_dbgmsg("SLOW IO: zio timestamp %lluns, "
                                    "delta %lluns, last io %lluns",
                                    fio->io_timestamp, delta,
                                    vq->vq_io_complete_ts);
                                fm_panic("I/O to pool ’%s’ appears to be "
                                    "hung.", spa_name(spa));
                        }
                }
                mutex_exit(&vq->vq_lock);
        }
}
```

```
**********************************************************
   58138 Wed May  6 08:47:28 2015
new/usr/src/uts/common/fs/zfs/zil.c
5269 zfs: zpool import slow
PORTING: this code relies on the property of taskq_wait to wait
until no more tasks are queued and no more tasks are active. As
we always queue new tasks from within other tasks, task_wait
reliably waits for the full recursion to finish, even though we
enqueue new tasks after taskq_wait has been called.
On platforms other than illumos, taskq_wait may not have this
property.
Reviewed by: Matthew Ahrens <mahrens@delphix.com>
Reviewed by: Dan McDonald <danmcd@omniti.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_
```

```
int
zil_claim(dsl_pool_t *dp, dsl_dataset_t *ds, void *txarg)
zil_claim(const char *osname, void *txarg)
{
        dmu_tx_t *tx = txarg;
        uint64_t first_txg = dmu_tx_get_txg(tx);
        zilog_t *zilog;
        zil_header_t *zh;
        objset_t *os;
        int error;

        error = dmu_objset_own_obj(dp, ds->ds_object,
            DMU_OST_ANY, B_FALSE, FTAG, &os);
        error = dmu_objset_own(osname, DMU_OST_ANY, B_FALSE, FTAG, &os);
        if (error != 0) {
                /*
                 * EBUSY indicates that the objset is inconsistent, in which
                 * case it can not have a ZIL.
                 */
                if (error != EBUSY) {
                        cmn_err(CE_WARN, "can’t open objset for %llu, error %u",
                            (unsigned long long)ds->ds_object, error);
                        cmn_err(CE_WARN, "can’t open objset for %s, error %u",
                            osname, error);
                }
                return (0);
        }

        zilog = dmu_objset_zil(os);
        zh = zil_header_in_syncing_context(zilog);

        if (spa_get_log_state(zilog->zl_spa) == SPA_LOG_CLEAR) {
                if (!BP_IS_HOLE(&zh->zh_log))
                        zio_free_zil(zilog->zl_spa, first_txg, &zh->zh_log);
                BP_ZERO(&zh->zh_log);
                dsl_dataset_dirty(dmu_objset_ds(os), tx);
                dmu_objset_disown(os, FTAG);
                return (0);
        }

        /*
         * Claim all log blocks if we haven’t already done so, and remember
         * the highest claimed sequence number.  This ensures that if we can
         * read only part of the log now (e.g. due to a missing device),
         * but we can read the entire log later, we will not try to replay
         * or destroy beyond the last block we successfully claimed.
         */
        ASSERT3U(zh->zh_claim_txg, <=, first_txg);
        if (zh->zh_claim_txg == 0 && !BP_IS_HOLE(&zh->zh_log)) {
```

```
                (void) zil_parse(zilog, zil_claim_log_block,
                    zil_claim_log_record, tx, first_txg);
                zh->zh_claim_txg = first_txg;
                zh->zh_claim_blk_seq = zilog->zl_parse_blk_seq;
                zh->zh_claim_lr_seq = zilog->zl_parse_lr_seq;
                if (zilog->zl_parse_lr_count || zilog->zl_parse_blk_count > 1)
                        zh->zh_flags |= ZIL_REPLAY_NEEDED;
                zh->zh_flags |= ZIL_CLAIM_LR_SEQ_VALID;
                dsl_dataset_dirty(dmu_objset_ds(os), tx);
        }

        ASSERT3U(first_txg, ==, (spa_last_synced_txg(zilog->zl_spa) + 1));
        dmu_objset_disown(os, FTAG);
        return (0);
}

/*
 * Check the log by walking the log chain.
 * Checksum errors are ok as they indicate the end of the chain.
 * Any other error (no device or read failure) returns an error.
 */
/* ARGSUSED */
#endif /* ! codereview */
int
zil_check_log_chain(dsl_pool_t *dp, dsl_dataset_t *ds, void *tx)
zil_check_log_chain(const char *osname, void *tx)
{
        zilog_t *zilog;
        objset_t *os;
        blkptr_t *bp;
        int error;

        ASSERT(tx == NULL);

        error = dmu_objset_from_ds(ds, &os);
        error = dmu_objset_hold(osname, FTAG, &os);
        if (error != 0) {
                cmn_err(CE_WARN, "can’t open objset %llu, error %d",
                    (unsigned long long)ds->ds_object, error);
                cmn_err(CE_WARN, "can’t open objset for %s", osname);
                return (0);
        }

        zilog = dmu_objset_zil(os);
        bp = (blkptr_t *)&zilog->zl_header->zh_log;

        /*
         * Check the first block and determine if it’s on a log device
         * which may have been removed or faulted prior to loading this
         * pool.  If so, there’s no point in checking the rest of the log
         * as its content should have already been synced to the pool.
         */
        if (!BP_IS_HOLE(bp)) {
                vdev_t *vd;
                boolean_t valid = B_TRUE;

                spa_config_enter(os->os_spa, SCL_STATE, FTAG, RW_READER);
                vd = vdev_lookup_top(os->os_spa, DVA_GET_VDEV(&bp->blk_dva[0]));
                if (vd->vdev_islog && vdev_is_dead(vd))
                        valid = vdev_log_state_valid(vd);
                spa_config_exit(os->os_spa, SCL_STATE, FTAG);

                if (!valid)
                if (!valid) {
                        dmu_objset_rele(os, FTAG);
                        return (0);
        }
        }

        /*
         * Because tx == NULL, zil_claim_log_block() will not actually claim
         * any blocks, but just determine whether it is possible to do so.
         * In addition to checking the log chain, zil_claim_log_block()
         * will invoke zio_claim() with a done func of spa_claim_notify(),
         * which will update spa_max_claim_txg.  See spa_load() for details.
         */
        error = zil_parse(zilog, zil_claim_log_block, zil_claim_log_record, tx,
            zilog->zl_header->zh_claim_txg ? -1ULL : spa_first_txg(os->os_spa));

        dmu_objset_rele(os, FTAG);

        return ((error == ECKSUM || error == ENOENT) ? 0 : error);
}
```

______***unchanged_portion_omitted***_