```
**********************************************************
   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)
{
```

```
118         return (os->os_phys->os_type);
119 }
120
121 void
122 dmu_objset_name(objset_t *os, char *buf)
123 {
124         dsl_dataset_name(os->os_dsl_dataset, buf);
125 }
126
127 uint64_t
128 dmu_objset_id(objset_t *os)
129 {
130         dsl_dataset_t *ds = os->os_dsl_dataset;
131
132         return (ds ? ds->ds_object : 0);
133 }
134
135 zfs_sync_type_t
136 dmu_objset_syncprop(objset_t *os)
137 {
138         return (os->os_sync);
139 }
140
141 zfs_logbias_op_t
142 dmu_objset_logbias(objset_t *os)
143 {
144         return (os->os_logbias);
145 }
146
147 static void
148 checksum_changed_cb(void *arg, uint64_t newval)
149 {
150         objset_t *os = arg;
151
152         /*
153          * Inheritance should have been done by now.
154          */
155         ASSERT(newval != ZIO_CHECKSUM_INHERIT);
156
157         os->os_checksum = zio_checksum_select(newval, ZIO_CHECKSUM_ON_VALUE);
158 }
159
160 static void
161 compression_changed_cb(void *arg, uint64_t newval)
162 {
163         objset_t *os = arg;
164
165         /*
166          * Inheritance and range checking should have been done by now.
167          */
168         ASSERT(newval != ZIO_COMPRESS_INHERIT);
169
170         os->os_compress = zio_compress_select(os->os_spa, newval,
171             ZIO_COMPRESS_ON);
172 }
173
174 static void
175 copies_changed_cb(void *arg, uint64_t newval)
176 {
177         objset_t *os = arg;
178
179         /*
180          * Inheritance and range checking should have been done by now.
181          */
182         ASSERT(newval > 0);
183         ASSERT(newval <= spa_max_replication(os->os_spa));
```



```
185         os->os_copies = newval;
186 }
187
188 static void
189 dedup_changed_cb(void *arg, uint64_t newval)
190 {
191         objset_t *os = arg;
192         spa_t *spa = os->os_spa;
193         enum zio_checksum checksum;
194
195         /*
196          * Inheritance should have been done by now.
197          */
198         ASSERT(newval != ZIO_CHECKSUM_INHERIT);
199
200         checksum = zio_checksum_dedup_select(spa, newval, ZIO_CHECKSUM_OFF);
201
202         os->os_dedup_checksum = checksum & ZIO_CHECKSUM_MASK;
203         os->os_dedup_verify = !!(checksum & ZIO_CHECKSUM_VERIFY);
204 }
205
206 static void
207 primary_cache_changed_cb(void *arg, uint64_t newval)
208 {
209         objset_t *os = arg;
210
211         /*
212          * Inheritance and range checking should have been done by now.
213          */
214         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
215             newval == ZFS_CACHE_METADATA);
216
217         os->os_primary_cache = newval;
218 }
219
220 static void
221 secondary_cache_changed_cb(void *arg, uint64_t newval)
222 {
223         objset_t *os = arg;
224
225         /*
226          * Inheritance and range checking should have been done by now.
227          */
228         ASSERT(newval == ZFS_CACHE_ALL || newval == ZFS_CACHE_NONE ||
229             newval == ZFS_CACHE_METADATA);
230
231         os->os_secondary_cache = newval;
232 }
233
234 static void
235 sync_changed_cb(void *arg, uint64_t newval)
236 {
237         objset_t *os = arg;
238
239         /*
240          * Inheritance and range checking should have been done by now.
241          */
242         ASSERT(newval == ZFS_SYNC_STANDARD || newval == ZFS_SYNC_ALWAYS ||
243             newval == ZFS_SYNC_DISABLED);
244
245         os->os_sync = newval;
246         if (os->os_zil)
247                 zil_set_sync(os->os_zil, newval);
248 }
```

```
250 static void
251 redundant_metadata_changed_cb(void *arg, uint64_t newval)
252 {
253         objset_t *os = arg;

255         /*
256          * Inheritance and range checking should have been done by now.
257          */
258         ASSERT(newval == ZFS_REDUNDANT_METADATA_ALL ||
259             newval == ZFS_REDUNDANT_METADATA_MOST);

261         os->os_redundant_metadata = newval;
262 }

264 static void
265 logbias_changed_cb(void *arg, uint64_t newval)
266 {
267         objset_t *os = arg;

269         ASSERT(newval == ZFS_LOGBIAS_LATENCY ||
270             newval == ZFS_LOGBIAS_THROUGHPUT);
271         os->os_logbias = newval;
272         if (os->os_zil)
273                 zil_set_logbias(os->os_zil, newval);
274 }

276 static void
277 recordsize_changed_cb(void *arg, uint64_t newval)
278 {
279         objset_t *os = arg;

281         os->os_recordsize = newval;
282 }

284 void
285 dmu_objset_byteswap(void *buf, size_t size)
286 {
287         objset_phys_t *osp = buf;

289         ASSERT(size == OBJSET_OLD_PHYS_SIZE || size == sizeof (objset_phys_t));
290         dnode_byteswap(&osp->os_meta_dnode);
291         byteswap_uint64_array(&osp->os_zil_header, sizeof (zil_header_t));
292         osp->os_type = BSWAP_64(osp->os_type);
293         osp->os_flags = BSWAP_64(osp->os_flags);
294         if (size == sizeof (objset_phys_t)) {
295                 dnode_byteswap(&osp->os_userused_dnode);
296                 dnode_byteswap(&osp->os_groupused_dnode);
297         }
298 }

300 int
301 dmu_objset_open_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
302     objset_t **osp)
303 {
304         objset_t *os;
305         int i, err;

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

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



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

319                 if (DMU_OS_IS_L2CACHEABLE(os))
320                         aflags |= ARC_FLAG_L2CACHE;
321                 if (DMU_OS_IS_L2COMPRESSIBLE(os))
322                         aflags |= ARC_FLAG_L2COMPRESS;

324                 dprintf_bp(os->os_rootbp, "reading %s", "");
325                 err = arc_read(NULL, spa, os->os_rootbp,
326                     arc_getbuf_func, &os->os_phys_buf,
327                     ZIO_PRIORITY_SYNC_READ, ZIO_FLAG_CANFAIL, &aflags, &zb);
328                 if (err != 0) {
329                         kmem_free(os, sizeof (objset_t));
330                         /* convert checksum errors into IO errors */
331                         if (err == ECKSUM)
332                                 err = SET_ERROR(EIO);
333                         return (err);
334                 }

336                 /* Increase the blocksize if we are permitted. */
337                 if (spa_version(spa) >= SPA_VERSION_USERSPACE &&
338                     arc_buf_size(os->os_phys_buf) < sizeof (objset_phys_t)) {
339                         arc_buf_t *buf = arc_buf_alloc(spa,
340                             sizeof (objset_phys_t), &os->os_phys_buf,
341                             ARC_BUFC_METADATA);
342                         bzero(buf->b_data, sizeof (objset_phys_t));
343                         bcopy(os->os_phys_buf->b_data, buf->b_data,
344                             arc_buf_size(os->os_phys_buf));
345                         (void) arc_buf_remove_ref(os->os_phys_buf,
346                             &os->os_phys_buf);
347                         os->os_phys_buf = buf;
348                 }

350                 os->os_phys = os->os_phys_buf->b_data;
351                 os->os_flags = os->os_phys->os_flags;
352         } else {
353                 int size = spa_version(spa) >= SPA_VERSION_USERSPACE ?
354                     sizeof (objset_phys_t) : OBJSET_OLD_PHYS_SIZE;
355                 os->os_phys_buf = arc_buf_alloc(spa, size,
356                     &os->os_phys_buf, ARC_BUFC_METADATA);
357                 os->os_phys = os->os_phys_buf->b_data;
358                 bzero(os->os_phys, size);
359         }

361         /*
362          * Note: the changed_cb will be called once before the register
363          * func returns, thus changing the checksum/compression from the
364          * default (fletcher2/off).  Snapshots don't need to know about
365          * checksum/compression/copies.
366          */
367         if (ds != NULL) {
368                 err = dsl_prop_register(ds,
369                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
370                     primary_cache_changed_cb, os);
371                 if (err == 0) {
372                         err = dsl_prop_register(ds,
373                             zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
374                             secondary_cache_changed_cb, os);
375                 }
376                 if (!ds->ds_is_snapshot) {
377                         if (err == 0) {
378                                 err = dsl_prop_register(ds,
379                                     zfs_prop_to_name(ZFS_PROP_CHECKSUM),
380                                     checksum_changed_cb, os);
381                         }
```

```
382                         if (err == 0) {
383                                 err = dsl_prop_register(ds,
384                                     zfs_prop_to_name(ZFS_PROP_COMPRESSION),
385                                     compression_changed_cb, os);
386                         }
387                         if (err == 0) {
388                                 err = dsl_prop_register(ds,
389                                     zfs_prop_to_name(ZFS_PROP_COPIES),
390                                     copies_changed_cb, os);
391                         }
392                         if (err == 0) {
393                                 err = dsl_prop_register(ds,
394                                     zfs_prop_to_name(ZFS_PROP_DEDUP),
395                                     dedup_changed_cb, os);
396                         }
397                         if (err == 0) {
398                                 err = dsl_prop_register(ds,
399                                     zfs_prop_to_name(ZFS_PROP_LOGBIAS),
400                                     logbias_changed_cb, os);
401                         }
402                         if (err == 0) {
403                                 err = dsl_prop_register(ds,
404                                     zfs_prop_to_name(ZFS_PROP_SYNC),
405                                     sync_changed_cb, os);
406                         }
407                         if (err == 0) {
408                                 err = dsl_prop_register(ds,
409                                     zfs_prop_to_name(
410                                     ZFS_PROP_REDUNDANT_METADATA),
411                                     redundant_metadata_changed_cb, os);
412                         }
413                         if (err == 0) {
414                                 err = dsl_prop_register(ds,
415                                     zfs_prop_to_name(ZFS_PROP_RECORDSIZE),
416                                     recordsize_changed_cb, os);
417                         }
418                 }
419                 if (err != 0) {
420                         VERIFY(arc_buf_remove_ref(os->os_phys_buf,
421                             &os->os_phys_buf));
422                         kmem_free(os, sizeof (objset_t));
423                         return (err);
424                 }
425         } else {
426                 /* It's the meta-objset. */
427                 os->os_checksum = ZIO_CHECKSUM_FLETCHER_4;
428                 os->os_compress = ZIO_COMPRESS_ON;
429                 os->os_copies = spa_max_replication(spa);
430                 os->os_dedup_checksum = ZIO_CHECKSUM_OFF;
431                 os->os_dedup_verify = B_FALSE;
432                 os->os_logbias = ZFS_LOGBIAS_LATENCY;
433                 os->os_sync = ZFS_SYNC_STANDARD;
434                 os->os_primary_cache = ZFS_CACHE_ALL;
435                 os->os_secondary_cache = ZFS_CACHE_ALL;
436         }

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

442         for (i = 0; i < TXG_SIZE; i++) {
443                 list_create(&os->os_dirty_dnodes[i], sizeof (dnode_t),
444                     offsetof(dnode_t, dn_dirty_link[i]));
445                 list_create(&os->os_free_dnodes[i], sizeof (dnode_t),
446                     offsetof(dnode_t, dn_dirty_link[i]));
447         }
```



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

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

457         dnode_special_open(os, &os->os_phys->os_meta_dnode,
458             DMU_META_DNODE_OBJECT, &os->os_meta_dnode);
459         if (arc_buf_size(os->os_phys_buf) >= sizeof (objset_phys_t)) {
460                 dnode_special_open(os, &os->os_phys->os_userused_dnode,
461                     DMU_USERUSED_OBJECT, &os->os_userused_dnode);
462                 dnode_special_open(os, &os->os_phys->os_groupused_dnode,
463                     DMU_GROUPUSED_OBJECT, &os->os_groupused_dnode);
464         }

466         *osp = os;
467         return (0);
468 }

470 int
471 dmu_objset_from_ds(dsl_dataset_t *ds, objset_t **osp)
472 {
473         int err = 0;

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

481                 if (err == 0) {
482                         mutex_enter(&ds->ds_lock);
483                         ASSERT(ds->ds_objset == NULL);
484                         ds->ds_objset = os;
485                         mutex_exit(&ds->ds_lock);
486                 }
487         }
488         *osp = ds->ds_objset;
489         mutex_exit(&ds->ds_opening_lock);
490         return (err);
491 }

493 /*
494  * Holds the pool while the objset is held.  Therefore only one objset
495  * can be held at a time.
496  */
497 int
498 dmu_objset_hold(const char *name, void *tag, objset_t **osp)
499 {
500         dsl_pool_t *dp;
501         dsl_dataset_t *ds;
502         int err;

504         err = dsl_pool_hold(name, tag, &dp);
505         if (err != 0)
506                 return (err);
507         err = dsl_dataset_hold(dp, name, tag, &ds);
508         if (err != 0) {
509                 dsl_pool_rele(dp, tag);
510                 return (err);
511         }

513         err = dmu_objset_from_ds(ds, osp);
```

```
514         if (err != 0) {
515                 dsl_dataset_rele(ds, tag);
516                 dsl_pool_rele(dp, tag);
517         }

519         return (err);
520 }

522 static int
523 dmu_objset_own_impl(dsl_dataset_t *ds, dmu_objset_type_t type,
524     boolean_t readonly, void *tag, objset_t **osp)
525 {
526         int err;

528         err = dmu_objset_from_ds(ds, osp);
529         if (err != 0) {
530                 dsl_dataset_disown(ds, tag);
531         } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
532                 dsl_dataset_disown(ds, tag);
533                 return (SET_ERROR(EINVAL));
534         } else if (!readonly && dsl_dataset_is_snapshot(ds)) {
535                 dsl_dataset_disown(ds, tag);
536                 return (SET_ERROR(EROFS));
537         }
538         return (err);
539 }

541 #endif /* ! codereview */
542 /*
543  * dsl_pool must not be held when this is called.
544  * Upon successful return, there will be a longhold on the dataset,
545  * and the dsl_pool will not be held.
546  */
547 int
548 dmu_objset_own(const char *name, dmu_objset_type_t type,
549     boolean_t readonly, void *tag, objset_t **osp)
550 {
551         dsl_pool_t *dp;
552         dsl_dataset_t *ds;
553         int err;

555         err = dsl_pool_hold(name, FTAG, &dp);
556         if (err != 0)
557                 return (err);
558         err = dsl_dataset_own(dp, name, tag, &ds);
559         if (err != 0) {
560                 dsl_pool_rele(dp, FTAG);
561                 return (err);
562         }
563         err = dmu_objset_own_impl(ds, type, readonly, tag, osp);
564         dsl_pool_rele(dp, FTAG);

566         return (err);
567 }
568 #endif /* ! codereview */

570 int
571 dmu_objset_own_obj(dsl_pool_t *dp, uint64_t obj, dmu_objset_type_t type,
572     boolean_t readonly, void *tag, objset_t **osp)
573 {
574         dsl_dataset_t *ds;
575         int err;

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



```
 28         err = dmu_objset_from_ds(ds, osp);
 29         dsl_pool_rele(dp, FTAG);
 30         if (err != 0) {
 31                 dsl_dataset_disown(ds, tag);
 32         } else if (type != DMU_OST_ANY && type != (*osp)->os_phys->os_type) {
 33                 dsl_dataset_disown(ds, tag);
 34                 return (SET_ERROR(EINVAL));
 35         } else if (!readonly && ds->ds_is_snapshot) {
 36                 dsl_dataset_disown(ds, tag);
 37                 return (SET_ERROR(EROFS));
 38         }
579                 return (err);

581         return (dmu_objset_own_impl(ds, type, readonly, tag, osp));
582 #endif /* ! codereview */
583 }

585 void
586 dmu_objset_rele(objset_t *os, void *tag)
587 {
588         dsl_pool_t *dp = dmu_objset_pool(os);
589         dsl_dataset_rele(os->os_dsl_dataset, tag);
590         dsl_pool_rele(dp, tag);
591 }

593 /*
594  * When we are called, os MUST refer to an objset associated with a dataset
595  * that is owned by 'tag'; that is, is held and long held by 'tag' and ds_owner
596  * == tag.  We will then release and reacquire ownership of the dataset while
597  * holding the pool config_rwlock to avoid intervening namespace or ownership
598  * changes may occur.
599  *
600  * This exists solely to accommodate zfs_ioc_userspace_upgrade()'s desire to
601  * release the hold on its dataset and acquire a new one on the dataset of the
602  * same name so that it can be partially torn down and reconstructed.
603  */
604 void
605 dmu_objset_refresh_ownership(objset_t *os, void *tag)
606 {
607         dsl_pool_t *dp;
608         dsl_dataset_t *ds, *newds;
609         char name[MAXNAMELEN];

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

616         dsl_dataset_name(ds, name);
617         dp = dmu_objset_pool(os);
618         dsl_pool_config_enter(dp, FTAG);
619         dmu_objset_disown(os, tag);
620         VERIFY0(dsl_dataset_own(dp, name, tag, &newds));
621         VERIFY3P(newds, ==, os->os_dsl_dataset);
622         dsl_pool_config_exit(dp, FTAG);
623 }

625 void
626 dmu_objset_disown(objset_t *os, void *tag)
627 {
628         dsl_dataset_disown(os->os_dsl_dataset, tag);
629 }

631 void
632 dmu_objset_evict_dbufs(objset_t *os)
633 {
634         dnode_t dn_marker;
```

```
635         dnode_t *dn;

637         mutex_enter(&os->os_lock);
638         dn = list_head(&os->os_dnodes);
639         while (dn != NULL) {
640                 /*
641                  * Skip dnodes without holds.  We have to do this dance
642                  * because dnode_add_ref() only works if there is already a
643                  * hold.  If the dnode has no holds, then it has no dbufs.
644                  */
645                 if (dnode_add_ref(dn, FTAG)) {
646                         list_insert_after(&os->os_dnodes, dn, &dn_marker);
647                         mutex_exit(&os->os_lock);

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

652                         mutex_enter(&os->os_lock);
653                         dn = list_next(&os->os_dnodes, &dn_marker);
654                         list_remove(&os->os_dnodes, &dn_marker);
655                 } else {
656                         dn = list_next(&os->os_dnodes, dn);
657                 }
658         }
659         mutex_exit(&os->os_lock);

661         if (DMU_USERUSED_DNODE(os) != NULL) {
662                 dnode_evict_dbufs(DMU_GROUPUSED_DNODE(os));
663                 dnode_evict_dbufs(DMU_USERUSED_DNODE(os));
664         }
665         dnode_evict_dbufs(DMU_META_DNODE(os));
666 }

668 /*
669  * Objset eviction processing is split into into two pieces.
670  * The first marks the objset as evicting, evicts any dbufs that
671  * have a refcount of zero, and then queues up the objset for the
672  * second phase of eviction.  Once os->os_dnodes has been cleared by
673  * dnode_buf_pageout()->dnode_destroy(), the second phase is executed.
674  * The second phase closes the special dnodes, dequeues the objset from
675  * the list of those undergoing eviction, and finally frees the objset.
676  *
677  * NOTE: Due to asynchronous eviction processing (invocation of
678  *       dnode_buf_pageout()), it is possible for the meta dnode for the
679  *       objset to have no holds even though os->os_dnodes is not empty.
680  */
681 void
682 dmu_objset_evict(objset_t *os)
683 {
684         dsl_dataset_t *ds = os->os_dsl_dataset;

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

689         if (ds) {
690                 if (!ds->ds_is_snapshot) {
691                         VERIFY0(dsl_prop_unregister(ds,
692                             zfs_prop_to_name(ZFS_PROP_CHECKSUM),
693                             checksum_changed_cb, os));
694                         VERIFY0(dsl_prop_unregister(ds,
695                             zfs_prop_to_name(ZFS_PROP_COMPRESSION),
696                             compression_changed_cb, os));
697                         VERIFY0(dsl_prop_unregister(ds,
698                             zfs_prop_to_name(ZFS_PROP_COPIES),
699                             copies_changed_cb, os));
700                         VERIFY0(dsl_prop_unregister(ds,
```



```
701                             zfs_prop_to_name(ZFS_PROP_DEDUP),
702                             dedup_changed_cb, os));
703                         VERIFY0(dsl_prop_unregister(ds,
704                             zfs_prop_to_name(ZFS_PROP_LOGBIAS),
705                             logbias_changed_cb, os));
706                         VERIFY0(dsl_prop_unregister(ds,
707                             zfs_prop_to_name(ZFS_PROP_SYNC),
708                             sync_changed_cb, os));
709                         VERIFY0(dsl_prop_unregister(ds,
710                             zfs_prop_to_name(ZFS_PROP_REDUNDANT_METADATA),
711                             redundant_metadata_changed_cb, os));
712                         VERIFY0(dsl_prop_unregister(ds,
713                             zfs_prop_to_name(ZFS_PROP_RECORDSIZE),
714                             recordsize_changed_cb, os));
715                 }
716                 VERIFY0(dsl_prop_unregister(ds,
717                     zfs_prop_to_name(ZFS_PROP_PRIMARYCACHE),
718                     primary_cache_changed_cb, os));
719                 VERIFY0(dsl_prop_unregister(ds,
720                     zfs_prop_to_name(ZFS_PROP_SECONDARYCACHE),
721                     secondary_cache_changed_cb, os));
722         }

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

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

730         mutex_enter(&os->os_lock);
731         spa_evicting_os_register(os->os_spa, os);
732         if (list_is_empty(&os->os_dnodes)) {
733                 mutex_exit(&os->os_lock);
734                 dmu_objset_evict_done(os);
735         } else {
736                 mutex_exit(&os->os_lock);
737         }
738 }

740 void
741 dmu_objset_evict_done(objset_t *os)
742 {
743         ASSERT3P(list_head(&os->os_dnodes), ==, NULL);

745         dnode_special_close(&os->os_meta_dnode);
746         if (DMU_USERUSED_DNODE(os)) {
747                 dnode_special_close(&os->os_userused_dnode);
748                 dnode_special_close(&os->os_groupused_dnode);
749         }
750         zil_free(os->os_zil);

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

754         /*
755          * This is a barrier to prevent the objset from going away in
756          * dnode_move() until we can safely ensure that the objset is still in
757          * use. We consider the objset valid before the barrier and invalid
758          * after the barrier.
759          */
760         rw_enter(&os_lock, RW_READER);
761         rw_exit(&os_lock);

763         mutex_destroy(&os->os_lock);
764         mutex_destroy(&os->os_obj_lock);
765         mutex_destroy(&os->os_user_ptr_lock);
766         spa_evicting_os_deregister(os->os_spa, os);
```

```
767         kmem_free(os, sizeof (objset_t));
768 }

770 timestruc_t
771 dmu_objset_snap_cmtime(objset_t *os)
772 {
773         return (dsl_dir_snap_cmtime(os->os_dsl_dataset->ds_dir));
774 }

776 /* called from dsl for meta-objset */
777 objset_t *
778 dmu_objset_create_impl(spa_t *spa, dsl_dataset_t *ds, blkptr_t *bp,
779     dmu_objset_type_t type, dmu_tx_t *tx)
780 {
781         objset_t *os;
782         dnode_t *mdn;

784         ASSERT(dmu_tx_is_syncing(tx));

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

791         mdn = DMU_META_DNODE(os);

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

796         /*
797          * We don't want to have to increase the meta-dnode's nlevels
798          * later, because then we could do it in quescing context while
799          * we are also accessing it in open context.
800          *
801          * This precaution is not necessary for the MOS (ds == NULL),
802          * because the MOS is only updated in syncing context.
803          * This is most fortunate: the MOS is the only objset that
804          * needs to be synced multiple times as spa_sync() iterates
805          * to convergence, so minimizing its dn_nlevels matters.
806          */
807         if (ds != NULL) {
808                 int levels = 1;

810                 /*
811                  * Determine the number of levels necessary for the meta-dnode
812                  * to contain DN_MAX_OBJECT dnodes.
813                  */
814                 while ((uint64_t)mdn->dn_nblkptr << (mdn->dn_datablkshift +
815                     (levels - 1) * (mdn->dn_indblkshift - SPA_BLKPTRSHIFT)) <
816                     DN_MAX_OBJECT * sizeof (dnode_phys_t))
817                         levels++;

819                 mdn->dn_next_nlevels[tx->tx_txg & TXG_MASK] =
820                     mdn->dn_nlevels = levels;
821         }

823         ASSERT(type != DMU_OST_NONE);
824         ASSERT(type != DMU_OST_ANY);
825         ASSERT(type < DMU_OST_NUMTYPES);
826         os->os_phys->os_type = type;
827         if (dmu_objset_userused_enabled(os)) {
828                 os->os_phys->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
829                 os->os_flags = os->os_phys->os_flags;
830         }

832         dsl_dataset_dirty(ds, tx);
```



```
834         return (os);
835 }

837 typedef struct dmu_objset_create_arg {
838         const char *doca_name;
839         cred_t *doca_cred;
840         void (*doca_userfunc)(objset_t *os, void *arg,
841             cred_t *cr, dmu_tx_t *tx);
842         void *doca_userarg;
843         dmu_objset_type_t doca_type;
844         uint64_t doca_flags;
845 } dmu_objset_create_arg_t;

847 /*ARGSUSED*/
848 static int
849 dmu_objset_create_check(void *arg, dmu_tx_t *tx)
850 {
851         dmu_objset_create_arg_t *doca = arg;
852         dsl_pool_t *dp = dmu_tx_pool(tx);
853         dsl_dir_t *pdd;
854         const char *tail;
855         int error;

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

860         error = dsl_dir_hold(dp, doca->doca_name, FTAG, &pdd, &tail);
861         if (error != 0)
862                 return (error);
863         if (tail == NULL) {
864                 dsl_dir_rele(pdd, FTAG);
865                 return (SET_ERROR(EEXIST));
866         }
867         error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
868             doca->doca_cred);
869         dsl_dir_rele(pdd, FTAG);

871         return (error);
872 }

874 static void
875 dmu_objset_create_sync(void *arg, dmu_tx_t *tx)
876 {
877         dmu_objset_create_arg_t *doca = arg;
878         dsl_pool_t *dp = dmu_tx_pool(tx);
879         dsl_dir_t *pdd;
880         const char *tail;
881         dsl_dataset_t *ds;
882         uint64_t obj;
883         blkptr_t *bp;
884         objset_t *os;

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

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

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

896         if (doca->doca_userfunc != NULL) {
897                 doca->doca_userfunc(os, doca->doca_userarg,
898                     doca->doca_cred, tx);
```

```
899         }

901         spa_history_log_internal_ds(ds, "create", tx, "");
902         dsl_dataset_rele(ds, FTAG);
903         dsl_dir_rele(pdd, FTAG);
904 }

906 int
907 dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
908     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg)
909 {
910         dmu_objset_create_arg_t doca;

912         doca.doca_name = name;
913         doca.doca_cred = CRED();
914         doca.doca_flags = flags;
915         doca.doca_userfunc = func;
916         doca.doca_userarg = arg;
917         doca.doca_type = type;

919         return (dsl_sync_task(name,
920             dmu_objset_create_check, dmu_objset_create_sync, &doca,
921             5, ZFS_SPACE_CHECK_NORMAL));
922 }

924 typedef struct dmu_objset_clone_arg {
925         const char *doca_clone;
926         const char *doca_origin;
927         cred_t *doca_cred;
928 } dmu_objset_clone_arg_t;

930 /*ARGSUSED*/
931 static int
932 dmu_objset_clone_check(void *arg, dmu_tx_t *tx)
933 {
934         dmu_objset_clone_arg_t *doca = arg;
935         dsl_dir_t *pdd;
936         const char *tail;
937         int error;
938         dsl_dataset_t *origin;
939         dsl_pool_t *dp = dmu_tx_pool(tx);

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

944         error = dsl_dir_hold(dp, doca->doca_clone, FTAG, &pdd, &tail);
945         if (error != 0)
946                 return (error);
947         if (tail == NULL) {
948                 dsl_dir_rele(pdd, FTAG);
949                 return (SET_ERROR(EEXIST));
950         }

952         error = dsl_fs_ss_limit_check(pdd, 1, ZFS_PROP_FILESYSTEM_LIMIT, NULL,
953             doca->doca_cred);
954         if (error != 0) {
955                 dsl_dir_rele(pdd, FTAG);
956                 return (SET_ERROR(EDQUOT));
957         }
958         dsl_dir_rele(pdd, FTAG);

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

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



```
965         if (!origin->ds_is_snapshot) {
966                 dsl_dataset_rele(origin, FTAG);
967                 return (SET_ERROR(EINVAL));
968         }
969         dsl_dataset_rele(origin, FTAG);

971         return (0);
972 }

974 static void
975 dmu_objset_clone_sync(void *arg, dmu_tx_t *tx)
976 {
977         dmu_objset_clone_arg_t *doca = arg;
978         dsl_pool_t *dp = dmu_tx_pool(tx);
979         dsl_dir_t *pdd;
980         const char *tail;
981         dsl_dataset_t *origin, *ds;
982         uint64_t obj;
983         char namebuf[MAXNAMELEN];

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

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

991         VERIFY0(dsl_dataset_hold_obj(pdd->dd_pool, obj, FTAG, &ds));
992         dsl_dataset_name(origin, namebuf);
993         spa_history_log_internal_ds(ds, "clone", tx,
994             "origin=%s (%llu)", namebuf, origin->ds_object);
995         dsl_dataset_rele(ds, FTAG);
996         dsl_dataset_rele(origin, FTAG);
997         dsl_dir_rele(pdd, FTAG);
998 }

1000 int
1001 dmu_objset_clone(const char *clone, const char *origin)
1002 {
1003         dmu_objset_clone_arg_t doca;

1005         doca.doca_clone = clone;
1006         doca.doca_origin = origin;
1007         doca.doca_cred = CRED();

1009         return (dsl_sync_task(clone,
1010             dmu_objset_clone_check, dmu_objset_clone_sync, &doca,
1011             5, ZFS_SPACE_CHECK_NORMAL));
1012 }

1014 int
1015 dmu_objset_snapshot_one(const char *fsname, const char *snapname)
1016 {
1017         int err;
1018         char *longsnap = kmem_asprintf("%s@%s", fsname, snapname);
1019         nvlist_t *snaps = fnvlist_alloc();

1021         fnvlist_add_boolean(snaps, longsnap);
1022         strfree(longsnap);
1023         err = dsl_dataset_snapshot(snaps, NULL, NULL);
1024         fnvlist_free(snaps);
1025         return (err);
1026 }

1028 static void
1029 dmu_objset_sync_dnodes(list_t *list, list_t *newlist, dmu_tx_t *tx)
1030 {
```

```
1031         dnode_t *dn;

1033         while (dn = list_head(list)) {
1034                 ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
1035                 ASSERT(dn->dn_dbuf->db_data_pending);
1036                 /*
1037                  * Initialize dn_zio outside dnode_sync() because the
1038                  * meta-dnode needs to set it ouside dnode_sync().
1039                  */
1040                 dn->dn_zio = dn->dn_dbuf->db_data_pending->dr_zio;
1041                 ASSERT(dn->dn_zio);

1043                 ASSERT3U(dn->dn_nlevels, <=, DN_MAX_LEVELS);
1044                 list_remove(list, dn);

1046                 if (newlist) {
1047                         (void) dnode_add_ref(dn, newlist);
1048                         list_insert_tail(newlist, dn);
1049                 }

1051                 dnode_sync(dn, tx);
1052         }
1053 }

1055 /* ARGSUSED */
1056 static void
1057 dmu_objset_write_ready(zio_t *zio, arc_buf_t *abuf, void *arg)
1058 {
1059         blkptr_t *bp = zio->io_bp;
1060         objset_t *os = arg;
1061         dnode_phys_t *dnp = &os->os_phys->os_meta_dnode;

1063         ASSERT(!BP_IS_EMBEDDED(bp));
1064         ASSERT3P(bp, ==, os->os_rootbp);
1065         ASSERT3U(BP_GET_TYPE(bp), ==, DMU_OT_OBJSET);
1066         ASSERT0(BP_GET_LEVEL(bp));

1068         /*
1069          * Update rootbp fill count: it should be the number of objects
1070          * allocated in the object set (not counting the "special"
1071          * objects that are stored in the objset_phys_t -- the meta
1072          * dnode and user/group accounting objects).
1073          */
1074         bp->blk_fill = 0;
1075         for (int i = 0; i < dnp->dn_nblkptr; i++)
1076                 bp->blk_fill += BP_GET_FILL(&dnp->dn_blkptr[i]);
1077 }

1079 /* ARGSUSED */
1080 static void
1081 dmu_objset_write_done(zio_t *zio, arc_buf_t *abuf, void *arg)
1082 {
1083         blkptr_t *bp = zio->io_bp;
1084         blkptr_t *bp_orig = &zio->io_bp_orig;
1085         objset_t *os = arg;

1087         if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
1088                 ASSERT(BP_EQUAL(bp, bp_orig));
1089         } else {
1090                 dsl_dataset_t *ds = os->os_dsl_dataset;
1091                 dmu_tx_t *tx = os->os_synctx;

1093                 (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
1094                 dsl_dataset_block_born(ds, bp, tx);
1095         }
1096 }
```



```
1098 /* called from dsl */
1099 void
1100 dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
1101 {
1102         int txgoff;
1103         zbookmark_phys_t zb;
1104         zio_prop_t zp;
1105         zio_t *zio;
1106         list_t *list;
1107         list_t *newlist = NULL;
1108         dbuf_dirty_record_t *dr;

1110         dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);

1112         ASSERT(dmu_tx_is_syncing(tx));
1113         /* XXX the write_done callback should really give us the tx... */
1114         os->os_synctx = tx;

1116         if (os->os_dsl_dataset == NULL) {
1117                 /*
1118                  * This is the MOS.  If we have upgraded,
1119                  * spa_max_replication() could change, so reset
1120                  * os_copies here.
1121                  */
1122                 os->os_copies = spa_max_replication(os->os_spa);
1123         }

1125         /*
1126          * Create the root block IO
1127          */
1128         SET_BOOKMARK(&zb, os->os_dsl_dataset ?
1129             os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
1130             ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1131         arc_release(os->os_phys_buf, &os->os_phys_buf);

1133         dmu_write_policy(os, NULL, 0, 0, &zp);

1135         zio = arc_write(pio, os->os_spa, tx->tx_txg,
1136             os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os),
1137             DMU_OS_IS_L2COMPRESSIBLE(os), &zp, dmu_objset_write_ready,
1138             NULL, dmu_objset_write_done, os, ZIO_PRIORITY_ASYNC_WRITE,
1139             ZIO_FLAG_MUSTSUCCEED, &zb);

1141         /*
1142          * Sync special dnodes - the parent IO for the sync is the root block
1143          */
1144         DMU_META_DNODE(os)->dn_zio = zio;
1145         dnode_sync(DMU_META_DNODE(os), tx);

1147         os->os_phys->os_flags = os->os_flags;

1149         if (DMU_USERUSED_DNODE(os) &&
1150             DMU_USERUSED_DNODE(os)->dn_type != DMU_OT_NONE) {
1151                 DMU_USERUSED_DNODE(os)->dn_zio = zio;
1152                 dnode_sync(DMU_USERUSED_DNODE(os), tx);
1153                 DMU_GROUPUSED_DNODE(os)->dn_zio = zio;
1154                 dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
1155         }

1157         txgoff = tx->tx_txg & TXG_MASK;

1159         if (dmu_objset_userused_enabled(os)) {
1160                 newlist = &os->os_synced_dnodes;
1161                 /*
1162                  * We must create the list here because it uses the
```

```
1163                  * dn_dirty_link[] of this txg.
1164                  */
1165                 list_create(newlist, sizeof (dnode_t),
1166                     offsetof(dnode_t, dn_dirty_link[txgoff]));
1167         }
1168
1169         dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
1170         dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);
1171
1172         list = &DMU_META_DNODE(os)->dn_dirty_records[txgoff];
1173         while (dr = list_head(list)) {
1174                 ASSERT0(dr->dr_dbuf->db_level);
1175                 list_remove(list, dr);
1176                 if (dr->dr_zio)
1177                         zio_nowait(dr->dr_zio);
1178         }
1179         /*
1180          * Free intent log blocks up to this tx.
1181          */
1182         zil_sync(os->os_zil, tx);
1183         os->os_phys->os_zil_header = os->os_zil_header;
1184         zio_nowait(zio);
1185 }
1186
1187 boolean_t
1188 dmu_objset_is_dirty(objset_t *os, uint64_t txg)
1189 {
1190         return (!list_is_empty(&os->os_dirty_dnodes[txg & TXG_MASK]) ||
1191             !list_is_empty(&os->os_free_dnodes[txg & TXG_MASK]));
1192 }
1193
1194 static objset_used_cb_t *used_cbs[DMU_OST_NUMTYPES];
1195
1196 void
1197 dmu_objset_register_type(dmu_objset_type_t ost, objset_used_cb_t *cb)
1198 {
1199         used_cbs[ost] = cb;
1200 }
1201
1202 boolean_t
1203 dmu_objset_userused_enabled(objset_t *os)
1204 {
1205         return (spa_version(os->os_spa) >= SPA_VERSION_USERSPACE &&
1206             used_cbs[os->os_phys->os_type] != NULL &&
1207             DMU_USERUSED_DNODE(os) != NULL);
1208 }
1209
1210 static void
1211 do_userquota_update(objset_t *os, uint64_t used, uint64_t flags,
1212     uint64_t user, uint64_t group, boolean_t subtract, dmu_tx_t *tx)
1213 {
1214         if ((flags & DNODE_FLAG_USERUSED_ACCOUNTED)) {
1215                 int64_t delta = DNODE_SIZE + used;
1216                 if (subtract)
1217                         delta = -delta;
1218                 VERIFY3U(0, ==, zap_increment_int(os, DMU_USERUSED_OBJECT,
1219                     user, delta, tx));
1220                 VERIFY3U(0, ==, zap_increment_int(os, DMU_GROUPUSED_OBJECT,
1221                     group, delta, tx));
1222         }
1223 }
1224
1225 void
1226 dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx)
1227 {
1228         dnode_t *dn;
```



```
1229         list_t *list = &os->os_synced_dnodes;
1230
1231         ASSERT(list_head(list) == NULL || dmu_objset_userused_enabled(os));
1232
1233         while (dn = list_head(list)) {
1234                 int flags;
1235                 ASSERT(!DMU_OBJECT_IS_SPECIAL(dn->dn_object));
1236                 ASSERT(dn->dn_phys->dn_type == DMU_OT_NONE ||
1237                     dn->dn_phys->dn_flags &
1238                     DNODE_FLAG_USERUSED_ACCOUNTED);
1239
1240                 /* Allocate the user/groupused objects if necessary. */
1241                 if (DMU_USERUSED_DNODE(os)->dn_type == DMU_OT_NONE) {
1242                         VERIFY(0 == zap_create_claim(os,
1243                             DMU_USERUSED_OBJECT,
1244                             DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1245                         VERIFY(0 == zap_create_claim(os,
1246                             DMU_GROUPUSED_OBJECT,
1247                             DMU_OT_USERGROUP_USED, DMU_OT_NONE, 0, tx));
1248                 }
1249
1250                 /*
1251                  * We intentionally modify the zap object even if the
1252                  * net delta is zero.  Otherwise
1253                  * the block of the zap obj could be shared between
1254                  * datasets but need to be different between them after
1255                  * a bprewrite.
1256                  */
1257
1258                 flags = dn->dn_id_flags;
1259                 ASSERT(flags);
1260                 if (flags & DN_ID_OLD_EXIST)  {
1261                         do_userquota_update(os, dn->dn_oldused, dn->dn_oldflags,
1262                             dn->dn_olduid, dn->dn_oldgid, B_TRUE, tx);
1263                 }
1264                 if (flags & DN_ID_NEW_EXIST) {
1265                         do_userquota_update(os, DN_USED_BYTES(dn->dn_phys),
1266                             dn->dn_phys->dn_flags,  dn->dn_newuid,
1267                             dn->dn_newgid, B_FALSE, tx);
1268                 }
1269
1270                 mutex_enter(&dn->dn_mtx);
1271                 dn->dn_oldused = 0;
1272                 dn->dn_oldflags = 0;
1273                 if (dn->dn_id_flags & DN_ID_NEW_EXIST) {
1274                         dn->dn_olduid = dn->dn_newuid;
1275                         dn->dn_oldgid = dn->dn_newgid;
1276                         dn->dn_id_flags |= DN_ID_OLD_EXIST;
1277                         if (dn->dn_bonuslen == 0)
1278                                 dn->dn_id_flags |= DN_ID_CHKED_SPILL;
1279                         else
1280                                 dn->dn_id_flags |= DN_ID_CHKED_BONUS;
1281                 }
1282                 dn->dn_id_flags &= ~(DN_ID_NEW_EXIST);
1283                 mutex_exit(&dn->dn_mtx);
1284
1285                 list_remove(list, dn);
1286                 dnode_rele(dn, list);
1287         }
1288 }
1289
1290 /*
1291  * Returns a pointer to data to find uid/gid from
1292  *
1293  * If a dirty record for transaction group that is syncing can't
1294  * 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;
        }
```

```
1427         mutex_exit(&dn->dn_mtx);
1428         if (have_spill)
1429                 dmu_buf_rele((dmu_buf_t *)db, FTAG);
1430 }

1432 boolean_t
1433 dmu_objset_userspace_present(objset_t *os)
1434 {
1435         return (os->os_phys->os_flags &
1436             OBJSET_FLAG_USERACCOUNTING_COMPLETE);
1437 }

1439 int
1440 dmu_objset_userspace_upgrade(objset_t *os)
1441 {
1442         uint64_t obj;
1443         int err = 0;

1445         if (dmu_objset_userspace_present(os))
1446                 return (0);
1447         if (!dmu_objset_userused_enabled(os))
1448                 return (SET_ERROR(ENOTSUP));
1449         if (dmu_objset_is_snapshot(os))
1450                 return (SET_ERROR(EINVAL));

1452         /*
1453          * We simply need to mark every object dirty, so that it will be
1454          * synced out and now accounted.  If this is called
1455          * concurrently, or if we already did some work before crashing,
1456          * that's fine, since we track each object's accounted state
1457          * independently.
1458          */

1460         for (obj = 0; err == 0; err = dmu_object_next(os, &obj, FALSE, 0)) {
1461                 dmu_tx_t *tx;
1462                 dmu_buf_t *db;
1463                 int objerr;

1465                 if (issig(JUSTLOOKING) && issig(FORREAL))
1466                         return (SET_ERROR(EINTR));

1468                 objerr = dmu_bonus_hold(os, obj, FTAG, &db);
1469                 if (objerr != 0)
1470                         continue;
1471                 tx = dmu_tx_create(os);
1472                 dmu_tx_hold_bonus(tx, obj);
1473                 objerr = dmu_tx_assign(tx, TXG_WAIT);
1474                 if (objerr != 0) {
1475                         dmu_tx_abort(tx);
1476                         continue;
1477                 }
1478                 dmu_buf_will_dirty(db, tx);
1479                 dmu_buf_rele(db, FTAG);
1480                 dmu_tx_commit(tx);
1481         }

1483         os->os_flags |= OBJSET_FLAG_USERACCOUNTING_COMPLETE;
1484         txg_wait_synced(dmu_objset_pool(os), 0);
1485         return (0);
1486 }

1488 void
1489 dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
1490     uint64_t *usedobjsp, uint64_t *availobjsp)
1491 {
1492         dsl_dataset_space(os->os_dsl_dataset, refdbytesp, availbytesp,
```



```
1493             usedobjsp, availobjsp);
1494 }

1496 uint64_t
1497 dmu_objset_fsid_guid(objset_t *os)
1498 {
1499         return (dsl_dataset_fsid_guid(os->os_dsl_dataset));
1500 }

1502 void
1503 dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat)
1504 {
1505         stat->dds_type = os->os_phys->os_type;
1506         if (os->os_dsl_dataset)
1507                 dsl_dataset_fast_stat(os->os_dsl_dataset, stat);
1508 }

1510 void
1511 dmu_objset_stats(objset_t *os, nvlist_t *nv)
1512 {
1513         ASSERT(os->os_dsl_dataset ||
1514             os->os_phys->os_type == DMU_OST_META);

1516         if (os->os_dsl_dataset != NULL)
1517                 dsl_dataset_stats(os->os_dsl_dataset, nv);

1519         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_TYPE,
1520             os->os_phys->os_type);
1521         dsl_prop_nvlist_add_uint64(nv, ZFS_PROP_USERACCOUNTING,
1522             dmu_objset_userspace_present(os));
1523 }

1525 int
1526 dmu_objset_is_snapshot(objset_t *os)
1527 {
1528         if (os->os_dsl_dataset != NULL)
1529                 return (os->os_dsl_dataset->ds_is_snapshot);
1530         else
1531                 return (B_FALSE);
1532 }

1534 int
1535 dmu_snapshot_realname(objset_t *os, char *name, char *real, int maxlen,
1536     boolean_t *conflict)
1537 {
1538         dsl_dataset_t *ds = os->os_dsl_dataset;
1539         uint64_t ignored;

1541         if (dsl_dataset_phys(ds)->ds_snapnames_zapobj == 0)
1542                 return (SET_ERROR(ENOENT));

1544         return (zap_lookup_norm(ds->ds_dir->dd_pool->dp_meta_objset,
1545             dsl_dataset_phys(ds)->ds_snapnames_zapobj, name, 8, 1, &ignored,
1546             MT_FIRST, real, maxlen, conflict));
1547 }

1549 int
1550 dmu_snapshot_list_next(objset_t *os, int namelen, char *name,
1551     uint64_t *idp, uint64_t *offp, boolean_t *case_conflict)
1552 {
1553         dsl_dataset_t *ds = os->os_dsl_dataset;
1554         zap_cursor_t cursor;
1555         zap_attribute_t attr;

1557         ASSERT(dsl_pool_config_held(dmu_objset_pool(os)));
```

```
1559         if (dsl_dataset_phys(ds)->ds_snapnames_zapobj == 0)
1560                 return (SET_ERROR(ENOENT));

1562         zap_cursor_init_serialized(&cursor,
1563             ds->ds_dir->dd_pool->dp_meta_objset,
1564             dsl_dataset_phys(ds)->ds_snapnames_zapobj, *offp);

1566         if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1567                 zap_cursor_fini(&cursor);
1568                 return (SET_ERROR(ENOENT));
1569         }

1571         if (strlen(attr.za_name) + 1 > namelen) {
1572                 zap_cursor_fini(&cursor);
1573                 return (SET_ERROR(ENAMETOOLONG));
1574         }

1576         (void) strcpy(name, attr.za_name);
1577         if (idp)
1578                 *idp = attr.za_first_integer;
1579         if (case_conflict)
1580                 *case_conflict = attr.za_normalization_conflict;
1581         zap_cursor_advance(&cursor);
1582         *offp = zap_cursor_serialize(&cursor);
1583         zap_cursor_fini(&cursor);

1585         return (0);
1586 }

1588 int
1589 dmu_dir_list_next(objset_t *os, int namelen, char *name,
1590     uint64_t *idp, uint64_t *offp)
1591 {
1592         dsl_dir_t *dd = os->os_dsl_dataset->ds_dir;
1593         zap_cursor_t cursor;
1594         zap_attribute_t attr;

1596         /* there is no next dir on a snapshot! */
1597         if (os->os_dsl_dataset->ds_object !=
1598             dsl_dir_phys(dd)->dd_head_dataset_obj)
1599                 return (SET_ERROR(ENOENT));

1601         zap_cursor_init_serialized(&cursor,
1602             dd->dd_pool->dp_meta_objset,
1603             dsl_dir_phys(dd)->dd_child_dir_zapobj, *offp);

1605         if (zap_cursor_retrieve(&cursor, &attr) != 0) {
1606                 zap_cursor_fini(&cursor);
1607                 return (SET_ERROR(ENOENT));
1608         }

1610         if (strlen(attr.za_name) + 1 > namelen) {
1611                 zap_cursor_fini(&cursor);
1612                 return (SET_ERROR(ENAMETOOLONG));
1613         }

1615         (void) strcpy(name, attr.za_name);
1616         if (idp)
1617                 *idp = attr.za_first_integer;
1618         zap_cursor_advance(&cursor);
1619         *offp = zap_cursor_serialize(&cursor);
1620         zap_cursor_fini(&cursor);

1622         return (0);
1623 }
```



```
1625 typedef struct dmu_objset_find_ctx {
1626         taskq_t         *dc_tq;
1627         dsl_pool_t      *dc_dp;
1628         uint64_t        dc_ddobj;
1629         int             (*dc_func)(dsl_pool_t *, dsl_dataset_t *, void *);
1630         void            *dc_arg;
1631         int             dc_flags;
1632         kmutex_t        *dc_error_lock;
1633         int             *dc_error;
1634 } dmu_objset_find_ctx_t;

1636 static void
1637 dmu_objset_find_dp_impl(dmu_objset_find_ctx_t *dcp)
  40 /*
  41  * Find objsets under and including ddobj, call func(ds) on each.
  42  */
  43 int
  44 dmu_objset_find_dp(dsl_pool_t *dp, uint64_t ddobj,
  45     int func(dsl_pool_t *, dsl_dataset_t *, void *), void *arg, int flags)
1638 {
1639         dsl_pool_t *dp = dcp->dc_dp;
1640         dmu_objset_find_ctx_t *child_dcp;
1641 #endif /* ! codereview */
1642         dsl_dir_t *dd;
1643         dsl_dataset_t *ds;
1644         zap_cursor_t zc;
1645         zap_attribute_t *attr;
1646         uint64_t thisobj;
1647         int err = 0;
  47         int err;

1649         /* don't process if there already was an error */
1650         if (*dcp->dc_error != 0)
1651                 goto out;
  49         ASSERT(dsl_pool_config_held(dp));

1653         err = dsl_dir_hold_obj(dp, dcp->dc_ddobj, NULL, FTAG, &dd);
  51         err = dsl_dir_hold_obj(dp, ddobj, NULL, FTAG, &dd);
1654         if (err != 0)
1655                 goto out;
  53                 return (err);

1657         /* Don't visit hidden ($MOS & $ORIGIN) objsets. */
1658         if (dd->dd_myname[0] == '$') {
1659                 dsl_dir_rele(dd, FTAG);
1660                 goto out;
  58                 return (0);
1661         }

1663         thisobj = dsl_dir_phys(dd)->dd_head_dataset_obj;
1664         attr = kmem_alloc(sizeof (zap_attribute_t), KM_SLEEP);

1666         /*
1667          * Iterate over all children.
1668          */
1669         if (dcp->dc_flags & DS_FIND_CHILDREN) {
  67         if (flags & DS_FIND_CHILDREN) {
1670                 for (zap_cursor_init(&zc, dp->dp_meta_objset,
1671                     dsl_dir_phys(dd)->dd_child_dir_zapobj);
1672                     zap_cursor_retrieve(&zc, attr) == 0;
1673                     (void) zap_cursor_advance(&zc)) {
1674                         ASSERT3U(attr->za_integer_length, ==,
1675                             sizeof (uint64_t));
1676                         ASSERT3U(attr->za_num_integers, ==, 1);

1678                         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
```

```
1796                  * of a writer lock. So we fall back to the synchronous path
1797                  * here.
1798                  * In the future it might be possible to get some magic into
1799                  * dsl_pool_config_held in a way that it returns true for
1800                  * the worker threads so that a single lock held from this
1801                  * thread suffices. For now, stay single threaded.
1802                  */
1803                 dmu_objset_find_dp_impl(dcp);

1805                 return (error);
1806         }

1808         ntasks = dmu_find_threads;
1809         if (ntasks == 0)
1810                 ntasks = vdev_count_leaves(dp->dp_spa) * 4;
1811         tq = taskq_create("dmu_objset_find", ntasks, minclsyspri, ntasks,
1812             INT_MAX, 0);
1813         if (tq == NULL) {
1814                 kmem_free(dcp, sizeof(*dcp));
1815                 return (SET_ERROR(ENOMEM));
1816         }
1817         dcp->dc_tq = tq;

1819         /* dcp will be freed by task */
1820         (void) taskq_dispatch(tq, dmu_objset_find_dp_cb, dcp, TQ_SLEEP);

1822         /*
1823          * PORTING: this code relies on the property of taskq_wait to wait
1824          * until no more tasks are queued and no more tasks are active. As
1825          * we always queue new tasks from within other tasks, task_wait
1826          * reliably waits for the full recursion to finish, even though we
1827          * enqueue new tasks after taskq_wait has been called.
1828          * On platforms other than illumos, taskq_wait may not have this
1829          * property.
1830          */
1831         taskq_wait(tq);
1832         taskq_destroy(tq);
1833         mutex_destroy(&err_lock);

1835         return (error);
 137         return (err);
1836 }
```

*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_

 754 void
 755 dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx)
 756 {
 757         ASSERT(dmu_tx_is_syncing(tx));
 758         ASSERT(dp->dp_origin_snap != NULL);

 760         VERIFY0(dmu_objset_find_dp(dp, dp->dp_root_dir_obj, upgrade_clones_cb,
 761             tx, DS_FIND_CHILDREN | DS_FIND_SERIALIZE));
 761             tx, DS_FIND_CHILDREN));
 762 }
______unchanged_portion_omitted_

 793 void
 794 dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx)
 795 {
 796         ASSERT(dmu_tx_is_syncing(tx));
 797         uint64_t obj;

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

 803         /*
 804          * We can't use bpobj_alloc(), because spa_version() still
 805          * returns the old version, and we need a new-version bpobj with
 806          * subobj support.  So call dmu_object_alloc() directly.
 807          */
 808         obj = dmu_object_alloc(dp->dp_meta_objset, DMU_OT_BPOBJ,
 809             SPA_OLD_MAXBLOCKSIZE, DMU_OT_BPOBJ_HDR, sizeof (bpobj_phys_t), tx);
 810         VERIFY0(zap_add(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 811             DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj, tx));
 812         VERIFY0(bpobj_open(&dp->dp_free_bpobj, dp->dp_meta_objset, obj));

 814         VERIFY0(dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
 815             upgrade_dir_clones_cb, tx, DS_FIND_CHILDREN | DS_FIND_SERIALIZE));
 815             upgrade_dir_clones_cb, tx, DS_FIND_CHILDREN));
 816 }
______unchanged_portion_omitted_

1060 boolean_t
1061 dsl_pool_config_held_writer(dsl_pool_t *dp)
1062 {
1063         return (RRW_WRITE_HELD(&dp->dp_config_rwlock));
1064 }
1065 #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_

1734 /*
1735  * Check for missing log devices
1736  */
1737 static boolean_t
1738 spa_check_logs(spa_t *spa)
1739 {
1740         boolean_t rv = B_FALSE;
1741         dsl_pool_t *dp = spa_get_dsl(spa);
1742 #endif /* ! codereview */

1744         switch (spa->spa_log_state) {
1745         case SPA_LOG_MISSING:
1746                 /* need to recheck in case slog has been restored */
1747         case SPA_LOG_UNKNOWN:
1748                 rv = (dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
1749                     zil_check_log_chain, NULL, DS_FIND_CHILDREN) != 0);
1741                 rv = (dmu_objset_find(spa->spa_name, zil_check_log_chain,
1742                     NULL, DS_FIND_CHILDREN) != 0);
1750                 if (rv)
1751                         spa_set_log_state(spa, SPA_LOG_MISSING);
1752                 break;
1753         }
1754         return (rv);
1755 }
______unchanged_portion_omitted_

2142 /*
2143  * Load an existing storage pool, using the pool's builtin spa_config as a
2144  * source of configuration information.
2145  */
2146 static int
2147 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
2148     spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
2149     char **ereport)
2150 {
2151         int error = 0;
2152         nvlist_t *nvroot = NULL;
2153         nvlist_t *label;
2154         vdev_t *rvd;
2155         uberblock_t *ub = &spa->spa_uberblock;
2156         uint64_t children, config_cache_txg = spa->spa_config_txg;
2157         int orig_mode = spa->spa_mode;
2158         int parse;
2159         uint64_t obj;
2160         boolean_t missing_feat_write = B_FALSE;

2162         /*
2163          * If this is an untrusted config, access the pool in read-only mode.
2164          * This prevents things like resilvering recently removed devices.
```



```
2165          */
2166         if (!mosconfig)
2167                 spa->spa_mode = FREAD;

2169         ASSERT(MUTEX_HELD(&spa_namespace_lock));

2171         spa->spa_load_state = state;

2173         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
2174                 return (SET_ERROR(EINVAL));

2176         parse = (type == SPA_IMPORT_EXISTING ?
2177             VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);

2179         /*
2180          * Create "The Godfather" zio to hold all async IOs
2181          */
2182         spa->spa_async_zio_root = kmem_alloc(max_ncpus * sizeof (void *),
2183             KM_SLEEP);
2184         for (int i = 0; i < max_ncpus; i++) {
2185                 spa->spa_async_zio_root[i] = zio_root(spa, NULL, NULL,
2186                     ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE |
2187                     ZIO_FLAG_GODFATHER);
2188         }

2190         /*
2191          * Parse the configuration into a vdev tree.  We explicitly set the
2192          * value that will be returned by spa_version() since parsing the
2193          * configuration requires knowing the version number.
2194          */
2195         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2196         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
2197         spa_config_exit(spa, SCL_ALL, FTAG);

2199         if (error != 0)
2200                 return (error);

2202         ASSERT(spa->spa_root_vdev == rvd);

2204         if (type != SPA_IMPORT_ASSEMBLE) {
2205                 ASSERT(spa_guid(spa) == pool_guid);
2206         }

2208         /*
2209          * Try to open all vdevs, loading each label in the process.
2210          */
2211         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2212         error = vdev_open(rvd);
2213         spa_config_exit(spa, SCL_ALL, FTAG);
2214         if (error != 0)
2215                 return (error);

2217         /*
2218          * We need to validate the vdev labels against the configuration that
2219          * we have in hand, which is dependent on the setting of mosconfig. If
2220          * mosconfig is true then we're validating the vdev labels based on
2221          * that config.  Otherwise, we're validating against the cached config
2222          * (zpool.cache) that was read when we loaded the zfs module, and then
2223          * later we will recursively call spa_load() and validate against
2224          * the vdev config.
2225          *
2226          * If we're assembling a new pool that's been split off from an
2227          * existing pool, the labels haven't yet been updated so we skip
2228          * validation for now.
2229          */
2230         if (type != SPA_IMPORT_ASSEMBLE) {
```

```
2231                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2232                error = vdev_validate(rvd, mosconfig);
2233                spa_config_exit(spa, SCL_ALL, FTAG);

2235                if (error != 0)
2236                        return (error);

2238                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2239                        return (SET_ERROR(ENXIO));
2240        }

2242        /*
2243         * Find the best uberblock.
2244         */
2245        vdev_uberblock_load(rvd, ub, &label);

2247        /*
2248         * If we weren't able to find a single valid uberblock, return failure.
2249         */
2250        if (ub->ub_txg == 0) {
2251                nvlist_free(label);
2252                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2253        }

2255        /*
2256         * If the pool has an unsupported version we can't open it.
2257         */
2258        if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2259                nvlist_free(label);
2260                return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2261        }

2263        if (ub->ub_version >= SPA_VERSION_FEATURES) {
2264                nvlist_t *features;

2266                /*
2267                 * If we weren't able to find what's necessary for reading the
2268                 * MOS in the label, return failure.
2269                 */
2270                if (label == NULL || nvlist_lookup_nvlist(label,
2271                    ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2272                        nvlist_free(label);
2273                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2274                            ENXIO));
2275                }

2277                /*
2278                 * Update our in-core representation with the definitive values
2279                 * from the label.
2280                 */
2281                nvlist_free(spa->spa_label_features);
2282                VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2283        }

2285        nvlist_free(label);

2287        /*
2288         * Look through entries in the label nvlist's features_for_read. If
2289         * there is a feature listed there which we don't understand then we
2290         * cannot open a pool.
2291         */
2292        if (ub->ub_version >= SPA_VERSION_FEATURES) {
2293                nvlist_t *unsup_feat;

2295                VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2296                    0);
```



```
2298                for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2299                    NULL); nvp != NULL;
2300                    nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2301                        if (!zfeature_is_supported(nvpair_name(nvp))) {
2302                                VERIFY(nvlist_add_string(unsup_feat,
2303                                    nvpair_name(nvp), "") == 0);
2304                        }
2305                }

2307                if (!nvlist_empty(unsup_feat)) {
2308                        VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2309                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2310                        nvlist_free(unsup_feat);
2311                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2312                            ENOTSUP));
2313                }

2315                nvlist_free(unsup_feat);
2316        }

2318        /*
2319         * If the vdev guid sum doesn't match the uberblock, we have an
2320         * incomplete configuration.  We first check to see if the pool
2321         * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2322         * If it is, defer the vdev_guid_sum check till later so we
2323         * can handle missing vdevs.
2324         */
2325        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2326            &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2327            rvd->vdev_guid_sum != ub->ub_guid_sum)
2328                return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));

2330        if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
2331                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2332                spa_try_repair(spa, config);
2333                spa_config_exit(spa, SCL_ALL, FTAG);
2334                nvlist_free(spa->spa_config_splitting);
2335                spa->spa_config_splitting = NULL;
2336        }

2338        /*
2339         * Initialize internal SPA structures.
2340         */
2341        spa->spa_state = POOL_STATE_ACTIVE;
2342        spa->spa_ubsync = spa->spa_uberblock;
2343        spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2344            TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2345        spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2346            spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2347        spa->spa_claim_max_txg = spa->spa_first_txg;
2348        spa->spa_prev_software_version = ub->ub_software_version;

2350        error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2351        if (error)
2352                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2353        spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;

2355        if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2356                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2358        if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2359                boolean_t missing_feat_read = B_FALSE;
2360                nvlist_t *unsup_feat, *enabled_feat;

2362                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
```

```
2363                    &spa->spa_feat_for_read_obj) != 0) {
2364                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2365                }

2367                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2368                    &spa->spa_feat_for_write_obj) != 0) {
2369                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2370                }

2372                if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
2373                    &spa->spa_feat_desc_obj) != 0) {
2374                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2375                }

2377                enabled_feat = fnvlist_alloc();
2378                unsup_feat = fnvlist_alloc();

2380                if (!spa_features_check(spa, B_FALSE,
2381                    unsup_feat, enabled_feat))
2382                        missing_feat_read = B_TRUE;

2384                if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2385                        if (!spa_features_check(spa, B_TRUE,
2386                            unsup_feat, enabled_feat)) {
2387                                missing_feat_write = B_TRUE;
2388                        }
2389                }

2391                fnvlist_add_nvlist(spa->spa_load_info,
2392                    ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);

2394                if (!nvlist_empty(unsup_feat)) {
2395                        fnvlist_add_nvlist(spa->spa_load_info,
2396                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
2397                }

2399                fnvlist_free(enabled_feat);
2400                fnvlist_free(unsup_feat);

2402                if (!missing_feat_read) {
2403                        fnvlist_add_boolean(spa->spa_load_info,
2404                            ZPOOL_CONFIG_CAN_RDONLY);
2405                }

2407                /*
2408                 * If the state is SPA_LOAD_TRYIMPORT, our objective is
2409                 * twofold: to determine whether the pool is available for
2410                 * import in read-write mode and (if it is not) whether the
2411                 * pool is available for import in read-only mode. If the pool
2412                 * is available for import in read-write mode, it is displayed
2413                 * as available in userland; if it is not available for import
2414                 * in read-only mode, it is displayed as unavailable in
2415                 * userland. If the pool is available for import in read-only
2416                 * mode but not read-write mode, it is displayed as unavailable
2417                 * in userland with a special note that the pool is actually
2418                 * available for open in read-only mode.
2419                 *
2420                 * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2421                 * missing a feature for write, we must first determine whether
2422                 * the pool can be opened read-only before returning to
2423                 * userland in order to know whether to display the
2424                 * abovementioned note.
2425                 */
2426                if (missing_feat_read || (missing_feat_write &&
2427                    spa_writeable(spa))) {
2428                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
```



```
2429                                ENOTSUP));
2430                }

2432                /*
2433                 * Load refcounts for ZFS features from disk into an in-memory
2434                 * cache during SPA initialization.
2435                 */
2436                for (spa_feature_t i = 0; i < SPA_FEATURES; i++) {
2437                        uint64_t refcount;

2439                        error = feature_get_refcount_from_disk(spa,
2440                            &spa_feature_table[i], &refcount);
2441                        if (error == 0) {
2442                                spa->spa_feat_refcount_cache[i] = refcount;
2443                        } else if (error == ENOTSUP) {
2444                                spa->spa_feat_refcount_cache[i] =
2445                                    SPA_FEATURE_DISABLED;
2446                        } else {
2447                                return (spa_vdev_err(rvd,
2448                                    VDEV_AUX_CORRUPT_DATA, EIO));
2449                        }
2450                }
2451        }

2453        if (spa_feature_is_active(spa, SPA_FEATURE_ENABLED_TXG)) {
2454                if (spa_dir_prop(spa, DMU_POOL_FEATURE_ENABLED_TXG,
2455                    &spa->spa_feat_enabled_txg_obj) != 0)
2456                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2457        }

2459        spa->spa_is_initializing = B_TRUE;
2460        error = dsl_pool_open(spa->spa_dsl_pool);
2461        spa->spa_is_initializing = B_FALSE;
2462        if (error != 0)
2463                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2465        if (!mosconfig) {
2466                uint64_t hostid;
2467                nvlist_t *policy = NULL, *nvconfig;

2469                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2470                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2472                if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2473                    ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2474                        char *hostname;
2475                        unsigned long myhostid = 0;

2477                        VERIFY(nvlist_lookup_string(nvconfig,
2478                            ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);

2480 #ifdef  _KERNEL
2481                        myhostid = zone_get_hostid(NULL);
2482 #else   /* _KERNEL */
2483                        /*
2484                         * We're emulating the system's hostid in userland, so
2485                         * we can't use zone_get_hostid().
2486                         */
2487                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
2488 #endif  /* _KERNEL */
2489                        if (hostid != 0 && myhostid != 0 &&
2490                            hostid != myhostid) {
2491                                nvlist_free(nvconfig);
2492                                cmn_err(CE_WARN, "pool '%s' could not be "
2493                                    "loaded as it was last accessed by "
2494                                    "another system (host: %s hostid: 0x%lx). "
```

```
2495                                    "See: http://illumos.org/msg/ZFS-8000-EY",
2496                                    spa_name(spa), hostname,
2497                                    (unsigned long)hostid);
2498                                return (SET_ERROR(EBADF));
2499                        }
2500                }
2501                if (nvlist_lookup_nvlist(spa->spa_config,
2502                    ZPOOL_REWIND_POLICY, &policy) == 0)
2503                        VERIFY(nvlist_add_nvlist(nvconfig,
2504                            ZPOOL_REWIND_POLICY, policy) == 0);

2506                spa_config_set(spa, nvconfig);
2507                spa_unload(spa);
2508                spa_deactivate(spa);
2509                spa_activate(spa, orig_mode);

2511                return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
2512        }

2514        if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
2515                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2516        error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
2517        if (error != 0)
2518                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2520        /*
2521         * Load the bit that tells us to use the new accounting function
2522         * (raid-z deflation).  If we have an older pool, this will not
2523         * be present.
2524         */
2525        error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
2526        if (error != 0 && error != ENOENT)
2527                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2529        error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
2530            &spa->spa_creation_version);
2531        if (error != 0 && error != ENOENT)
2532                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2534        /*
2535         * Load the persistent error log.  If we have an older pool, this will
2536         * not be present.
2537         */
2538        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
2539        if (error != 0 && error != ENOENT)
2540                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2542        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
2543            &spa->spa_errlog_scrub);
2544        if (error != 0 && error != ENOENT)
2545                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2547        /*
2548         * Load the history object.  If we have an older pool, this
2549         * will not be present.
2550         */
2551        error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
2552        if (error != 0 && error != ENOENT)
2553                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2555        /*
2556         * If we're assembling the pool from the split-off vdevs of
2557         * an existing pool, we don't want to attach the spares & cache
2558         * devices.
2559         */
```



```
2561        /*
2562         * Load any hot spares for this pool.
2563         */
2564        error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
2565        if (error != 0 && error != ENOENT)
2566                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2567        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2568                ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
2569                if (load_nvlist(spa, spa->spa_spares.sav_object,
2570                    &spa->spa_spares.sav_config) != 0)
2571                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2573                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2574                spa_load_spares(spa);
2575                spa_config_exit(spa, SCL_ALL, FTAG);
2576        } else if (error == 0) {
2577                spa->spa_spares.sav_sync = B_TRUE;
2578        }

2580        /*
2581         * Load any level 2 ARC devices for this pool.
2582         */
2583        error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
2584            &spa->spa_l2cache.sav_object);
2585        if (error != 0 && error != ENOENT)
2586                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2587        if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2588                ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
2589                if (load_nvlist(spa, spa->spa_l2cache.sav_object,
2590                    &spa->spa_l2cache.sav_config) != 0)
2591                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2593                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2594                spa_load_l2cache(spa);
2595                spa_config_exit(spa, SCL_ALL, FTAG);
2596        } else if (error == 0) {
2597                spa->spa_l2cache.sav_sync = B_TRUE;
2598        }

2600        spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);

2602        error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
2603        if (error && error != ENOENT)
2604                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2606        if (error == 0) {
2607                uint64_t autoreplace;

2609                spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
2610                spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
2611                spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
2612                spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2613                spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2614                spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2615                    &spa->spa_dedup_ditto);

2617                spa->spa_autoreplace = (autoreplace != 0);
2618        }

2620        /*
2621         * If the 'autoreplace' property is set, then post a resource notifying
2622         * the ZFS DE that it should not issue any faults for unopenable
2623         * devices.  We also iterate over the vdevs, and post a sysevent for any
2624         * unopenable vdevs so that the normal autoreplace handler can take
2625         * over.
2626         */
```

```
2627         if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2628                 spa_check_removed(spa->spa_root_vdev);
2629                 /*
2630                  * For the import case, this is done in spa_import(), because
2631                  * at this point we're using the spare definitions from
2632                  * the MOS config, not necessarily from the userland config.
2633                  */
2634                 if (state != SPA_LOAD_IMPORT) {
2635                         spa_aux_check_removed(&spa->spa_spares);
2636                         spa_aux_check_removed(&spa->spa_l2cache);
2637                 }
2638         }

2640         /*
2641          * Load the vdev state for all toplevel vdevs.
2642          */
2643         vdev_load(rvd);

2645         /*
2646          * Propagate the leaf DTLs we just loaded all the way up the tree.
2647          */
2648         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2649         vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2650         spa_config_exit(spa, SCL_ALL, FTAG);

2652         /*
2653          * Load the DDTs (dedup tables).
2654          */
2655         error = ddt_load(spa);
2656         if (error != 0)
2657                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2659         spa_update_dspace(spa);

2661         /*
2662          * Validate the config, using the MOS config to fill in any
2663          * information which might be missing.  If we fail to validate
2664          * the config then declare the pool unfit for use. If we're
2665          * assembling a pool from a split, the log is not transferred
2666          * over.
2667          */
2668         if (type != SPA_IMPORT_ASSEMBLE) {
2669                 nvlist_t *nvconfig;

2671                 if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2672                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2674                 if (!spa_config_valid(spa, nvconfig)) {
2675                         nvlist_free(nvconfig);
2676                         return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2677                             ENXIO));
2678                 }
2679                 nvlist_free(nvconfig);

2681                 /*
2682                  * Now that we've validated the config, check the state of the
2683                  * root vdev.  If it can't be opened, it indicates one or
2684                  * more toplevel vdevs are faulted.
2685                  */
2686                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2687                         return (SET_ERROR(ENXIO));

2689                 if (spa_check_logs(spa)) {
2690                         *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2691                         return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2692                 }
```



```
2693         }

2695         if (missing_feat_write) {
2696                 ASSERT(state == SPA_LOAD_TRYIMPORT);

2698                 /*
2699                  * At this point, we know that we can open the pool in
2700                  * read-only mode but not read-write mode. We now have enough
2701                  * information and can return to userland.
2702                  */
2703                 return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2704         }

2706         /*
2707          * We've successfully opened the pool, verify that we're ready
2708          * to start pushing transactions.
2709          */
2710         if (state != SPA_LOAD_TRYIMPORT) {
2711                 if (error = spa_load_verify(spa))
2712                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2713                             error));
2714         }

2716         if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2717             spa->spa_load_max_txg == UINT64_MAX)) {
2718                 dmu_tx_t *tx;
2719                 int need_update = B_FALSE;
2720                 dsl_pool_t *dp = spa_get_dsl(spa);
2721 #endif /* ! codereview */

2723                 ASSERT(state != SPA_LOAD_TRYIMPORT);

2725                 /*
2726                  * Claim log blocks that haven't been committed yet.
2727                  * This must all happen in a single txg.
2728                  * Note: spa_claim_max_txg is updated by spa_claim_notify(),
2729                  * invoked from zil_claim_log_block()'s i/o done callback.
2730                  * Price of rollback is that we abandon the log.
2731                  */
2732                 spa->spa_claiming = B_TRUE;

2734                 tx = dmu_tx_create_assigned(dp, spa_first_txg(spa));
2735                 (void) dmu_objset_find_dp(dp, dp->dp_root_dir_obj,
2713                 tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2714                     spa_first_txg(spa));
2715                 (void) dmu_objset_find(spa_name(spa),
2736                     zil_claim, tx, DS_FIND_CHILDREN);
2737                 dmu_tx_commit(tx);

2739                 spa->spa_claiming = B_FALSE;

2741                 spa_set_log_state(spa, SPA_LOG_GOOD);
2742                 spa->spa_sync_on = B_TRUE;
2743                 txg_sync_start(spa->spa_dsl_pool);

2745                 /*
2746                  * Wait for all claims to sync.  We sync up to the highest
2747                  * claimed log block birth time so that claimed log blocks
2748                  * don't appear to be from the future.  spa_claim_max_txg
2749                  * will have been set for us by either zil_check_log_chain()
2750                  * (invoked from spa_check_logs()) or zil_claim() above.
2751                  */
2752                 txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);

2754                 /*
2755                  * If the config cache is stale, or we have uninitialized
```

```
2756                  * metaslabs (see spa_vdev_add()), then update the config.
2757                  *
2758                  * If this is a verbatim import, trust the current
2759                  * in-core spa_config and update the disk labels.
2760                  */
2761                 if (config_cache_txg != spa->spa_config_txg ||
2762                     state == SPA_LOAD_IMPORT ||
2763                     state == SPA_LOAD_RECOVER ||
2764                     (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2765                         need_update = B_TRUE;

2767                 for (int c = 0; c < rvd->vdev_children; c++)
2768                         if (rvd->vdev_child[c]->vdev_ms_array == 0)
2769                                 need_update = B_TRUE;

2771                 /*
2772                  * Update the config cache asychronously in case we're the
2773                  * root pool, in which case the config cache isn't writable yet.
2774                  */
2775                 if (need_update)
2776                         spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2778                 /*
2779                  * Check all DTLs to see if anything needs resilvering.
2780                  */
2781                 if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2782                     vdev_resilver_needed(rvd, NULL, NULL))
2783                         spa_async_request(spa, SPA_ASYNC_RESILVER);

2785                 /*
2786                  * Log the fact that we booted up (so that we can detect if
2787                  * we rebooted in the middle of an operation).
2788                  */
2789                 spa_history_log_version(spa, "open");

2791                 /*
2792                  * Delete any inconsistent datasets.
2793                  */
2794                 (void) dmu_objset_find(spa_name(spa),
2795                     dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2797                 /*
2798                  * Clean up any stale temporary dataset userrefs.
2799                  */
2800                 dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2801         }

2803         return (0);
2804 }
```

_____***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_

 235 void byteswap_uint64_array(void *buf, size_t size);
 236 void byteswap_uint32_array(void *buf, size_t size);
 237 void byteswap_uint16_array(void *buf, size_t size);
 238 void byteswap_uint8_array(void *buf, size_t size);
 239 void zap_byteswap(void *buf, size_t size);
 240 void zfs_oldacl_byteswap(void *buf, size_t size);
 241 void zfs_acl_byteswap(void *buf, size_t size);
 242 void zfs_znode_byteswap(void *buf, size_t size);

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

 249 /*
 250  * The maximum number of bytes that can be accessed as part of one
 251  * operation, including metadata.
 252  */
 253 #define DMU_MAX_ACCESS (32 * 1024 * 1024) /* 32MB */
 254 #define DMU_MAX_DELETEBLKCNT (20480) /* ~5MB of indirect blocks */

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

 259 /*
 260  * artificial blkids for bonus buffer and spill blocks
 261  */
 262 #define DMU_BONUS_BLKID         (-1ULL)
 263 #define DMU_SPILL_BLKID         (-2ULL)
 264 /*
 265  * Public routines to create, destroy, open, and close objsets.
 266  */
 267 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
 268 int dmu_objset_own(const char *name, dmu_objset_type_t type,
 269     boolean_t readonly, void *tag, objset_t **osp);
 270 void dmu_objset_rele(objset_t *os, void *tag);
 271 void dmu_objset_disown(objset_t *os, void *tag);
 272 int dmu_objset_open_ds(struct dsl_dataset *ds, objset_t **osp);

 274 void dmu_objset_evict_dbufs(objset_t *os);
 275 int dmu_objset_create(const char *name, dmu_objset_type_t type, uint64_t flags,
 276     void (*func)(objset_t *os, void *arg, cred_t *cr, dmu_tx_t *tx), void *arg);
 277 int dmu_objset_clone(const char *name, const char *origin);
 278 int dsl_destroy_snapshots_nvl(struct nvlist *snaps, boolean_t defer,
 279     struct nvlist *errlist);
 280 int dmu_objset_snapshot_one(const char *fsname, const char *snapname);
 281 int dmu_objset_snapshot_tmp(const char *, const char *, int);
 282 int dmu_objset_find(char *name, int func(const char *, void *), void *arg,
 283     int flags);
```



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

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

 295 /*
 296  * The names of zap entries in the DIRECTORY_OBJECT of the MOS.
 297  */
 298 #define DMU_POOL_DIRECTORY_OBJECT       1
 299 #define DMU_POOL_CONFIG                 "config"
 300 #define DMU_POOL_FEATURES_FOR_WRITE     "features_for_write"
 301 #define DMU_POOL_FEATURES_FOR_READ      "features_for_read"
 302 #define DMU_POOL_FEATURE_DESCRIPTIONS   "feature_descriptions"
 303 #define DMU_POOL_FEATURE_ENABLED_TXG    "feature_enabled_txg"
 304 #define DMU_POOL_ROOT_DATASET           "root_dataset"
 305 #define DMU_POOL_SYNC_BPOBJ             "sync_bplist"
 306 #define DMU_POOL_ERRLOG_SCRUB           "errlog_scrub"
 307 #define DMU_POOL_ERRLOG_LAST            "errlog_last"
 308 #define DMU_POOL_SPARES                 "spares"
 309 #define DMU_POOL_DEFLATE                "deflate"
 310 #define DMU_POOL_HISTORY                "history"
 311 #define DMU_POOL_PROPS                  "pool_props"
 312 #define DMU_POOL_L2CACHE                "l2cache"
 313 #define DMU_POOL_TMP_USERREFS           "tmp_userrefs"
 314 #define DMU_POOL_DDT                    "DDT-%s-%s-%s"
 315 #define DMU_POOL_DDT_STATS              "DDT-statistics"
 316 #define DMU_POOL_CREATION_VERSION       "creation_version"
 317 #define DMU_POOL_SCAN                   "scan"
 318 #define DMU_POOL_FREE_BPOBJ             "free_bpobj"
 319 #define DMU_POOL_BPTREE_OBJ             "bptree_obj"
 320 #define DMU_POOL_EMPTY_BPOBJ            "empty_bpobj"

 322 /*
 323  * Allocate an object from this objset.  The range of object numbers
 324  * available is (0, DN_MAX_OBJECT).  Object 0 is the meta-dnode.
 325  *
 326  * The transaction must be assigned to a txg.  The newly allocated
 327  * object will be "held" in the transaction (ie. you can modify the
 328  * newly allocated object in this transaction).
 329  *
 330  * dmu_object_alloc() chooses an object and returns it in *objectp.
 331  *
 332  * dmu_object_claim() allocates a specific object number.  If that
 333  * number is already allocated, it fails and returns EEXIST.
 334  *
 335  * Return 0 on success, or ENOSPC or EEXIST as specified above.
 336  */
 337 uint64_t dmu_object_alloc(objset_t *os, dmu_object_type_t ot,
 338     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 339 int dmu_object_claim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 340     int blocksize, dmu_object_type_t bonus_type, int bonus_len, dmu_tx_t *tx);
 341 int dmu_object_reclaim(objset_t *os, uint64_t object, dmu_object_type_t ot,
 342     int blocksize, dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *txp);

 344 /*
 345  * Free an object from this objset.
 346  *
 347  * The object's data will be freed as well (ie. you don't need to call
 348  * dmu_free(object, 0, -1, tx)).
 349  *
```

```
 * 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);
```

```
482 /*
483  * dmu_buf_hold_array holds the DMU buffers which contain all bytes in a
484  * range of an object.  A pointer to an array of dmu_buf_t*'s is
485  * returned (in *dbpp).
486  *
487  * dmu_buf_rele_array releases the hold on an array of dmu_buf_t*'s, and
488  * frees the array.  The hold on the array of buffers MUST be released
489  * with dmu_buf_rele_array.  You can NOT release the hold on each buffer
490  * individually with dmu_buf_rele.
491  */
492 int dmu_buf_hold_array_by_bonus(dmu_buf_t *db, uint64_t offset,
493     uint64_t length, int read, void *tag, int *numbufsp, dmu_buf_t ***dbpp);
494 void dmu_buf_rele_array(dmu_buf_t **, int numbufs, void *tag);

496 typedef void dmu_buf_evict_func_t(void *user_ptr);

498 /*
499  * A DMU buffer user object may be associated with a dbuf for the
500  * duration of its lifetime.  This allows the user of a dbuf (client)
501  * to attach private data to a dbuf (e.g. in-core only data such as a
502  * dnode_children_t, zap_t, or zap_leaf_t) and be optionally notified
503  * when that dbuf has been evicted.  Clients typically respond to the
504  * eviction notification by freeing their private data, thus ensuring
505  * the same lifetime for both dbuf and private data.
506  *
507  * The mapping from a dmu_buf_user_t to any client private data is the
508  * client's responsibility.  All current consumers of the API with private
509  * data embed a dmu_buf_user_t as the first member of the structure for
510  * their private data.  This allows conversions between the two types
511  * with a simple cast.  Since the DMU buf user API never needs access
512  * to the private data, other strategies can be employed if necessary
513  * or convenient for the client (e.g. using container_of() to do the
514  * conversion for private data that cannot have the dmu_buf_user_t as
515  * its first member).
516  *
517  * Eviction callbacks are executed without the dbuf mutex held or any
518  * other type of mechanism to guarantee that the dbuf is still available.
519  * For this reason, users must assume the dbuf has already been freed
520  * and not reference the dbuf from the callback context.
521  *
522  * Users requesting "immediate eviction" are notified as soon as the dbuf
523  * is only referenced by dirty records (dirties == holds).  Otherwise the
524  * notification occurs after eviction processing for the dbuf begins.
525  */
526 typedef struct dmu_buf_user {
527         /*
528          * Asynchronous user eviction callback state.
529          */
530         taskq_ent_t     dbu_tqent;

532         /* This instance's eviction function pointer. */
533         dmu_buf_evict_func_t *dbu_evict_func;
534 #ifdef ZFS_DEBUG
535         /*
536          * Pointer to user's dbuf pointer.  NULL for clients that do
537          * not associate a dbuf with their user data.
538          *
539          * The dbuf pointer is cleared upon eviction so as to catch
540          * use-after-evict bugs in clients.
541          */
542         dmu_buf_t **dbu_clear_on_evict_dbufp;
543 #endif
544 } dmu_buf_user_t;

546 /*
547  * Initialize the given dmu_buf_user_t instance with the eviction function
```



```
548  * evict_func, to be called when the user is evicted.
549  *
550  * NOTE: This function should only be called once on a given dmu_buf_user_t.
551  *       To allow enforcement of this, dbu must already be zeroed on entry.
552  */
553 #ifdef __lint
554 /* Very ugly, but it beats issuing suppression directives in many Makefiles. */
555 extern void
556 dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
557     dmu_buf_t **clear_on_evict_dbufp);
558 #else /* __lint */
559 inline void
560 dmu_buf_init_user(dmu_buf_user_t *dbu, dmu_buf_evict_func_t *evict_func,
561     dmu_buf_t **clear_on_evict_dbufp)
562 {
563         ASSERT(dbu->dbu_evict_func == NULL);
564         ASSERT(evict_func != NULL);
565         dbu->dbu_evict_func = evict_func;
566 #ifdef ZFS_DEBUG
567         dbu->dbu_clear_on_evict_dbufp = clear_on_evict_dbufp;
568 #endif
569 }
570 #endif /* __lint */

572 /*
573  * Attach user data to a dbuf and mark it for normal (when the dbuf's
574  * data is cleared or its reference count goes to zero) eviction processing.
575  *
576  * Returns NULL on success, or the existing user if another user currently
577  * owns the buffer.
578  */
579 void *dmu_buf_set_user(dmu_buf_t *db, dmu_buf_user_t *user);

581 /*
582  * Attach user data to a dbuf and mark it for immediate (its dirty and
583  * reference counts are equal) eviction processing.
584  *
585  * Returns NULL on success, or the existing user if another user currently
586  * owns the buffer.
587  */
588 void *dmu_buf_set_user_ie(dmu_buf_t *db, dmu_buf_user_t *user);

590 /*
591  * Replace the current user of a dbuf.
592  *
593  * If given the current user of a dbuf, replaces the dbuf's user with
594  * "new_user" and returns the user data pointer that was replaced.
595  * Otherwise returns the current, and unmodified, dbuf user pointer.
596  */
597 void *dmu_buf_replace_user(dmu_buf_t *db,
598     dmu_buf_user_t *old_user, dmu_buf_user_t *new_user);

600 /*
601  * Remove the specified user data for a DMU buffer.
602  *
603  * Returns the user that was removed on success, or the current user if
604  * another user currently owns the buffer.
605  */
606 void *dmu_buf_remove_user(dmu_buf_t *db, dmu_buf_user_t *user);

608 /*
609  * Returns the user data (dmu_buf_user_t *) associated with this dbuf.
610  */
611 void *dmu_buf_get_user(dmu_buf_t *db);

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

```
614 void dmu_buf_user_evict_wait(void);

616 /*
617  * Returns the blkptr associated with this dbuf, or NULL if not set.
618  */
619 struct blkptr *dmu_buf_get_blkptr(dmu_buf_t *db);

621 /*
622  * Indicate that you are going to modify the buffer's data (db_data).
623  *
624  * The transaction (tx) must be assigned to a txg (ie. you've called
625  * dmu_tx_assign()).  The buffer's object must be held in the tx
626  * (ie. you've called dmu_tx_hold_object(tx, db->db_object)).
627  */
628 void dmu_buf_will_dirty(dmu_buf_t *db, dmu_tx_t *tx);

630 /*
631  * Tells if the given dbuf is freeable.
632  */
633 boolean_t dmu_buf_freeable(dmu_buf_t *);

635 /*
636  * You must create a transaction, then hold the objects which you will
637  * (or might) modify as part of this transaction.  Then you must assign
638  * the transaction to a transaction group.  Once the transaction has
639  * been assigned, you can modify buffers which belong to held objects as
640  * part of this transaction.  You can't modify buffers before the
641  * transaction has been assigned; you can't modify buffers which don't
642  * belong to objects which this transaction holds; you can't hold
643  * objects once the transaction has been assigned.  You may hold an
644  * object which you are going to free (with dmu_object_free()), but you
645  * don't have to.
646  *
647  * You can abort the transaction before it has been assigned.
648  *
649  * Note that you may hold buffers (with dmu_buf_hold) at any time,
650  * regardless of transaction state.
651  */

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

656 dmu_tx_t *dmu_tx_create(objset_t *os);
657 void dmu_tx_hold_write(dmu_tx_t *tx, uint64_t object, uint64_t off, int len);
658 void dmu_tx_hold_free(dmu_tx_t *tx, uint64_t object, uint64_t off,
659     uint64_t len);
660 void dmu_tx_hold_zap(dmu_tx_t *tx, uint64_t object, int add, const char *name);
661 void dmu_tx_hold_bonus(dmu_tx_t *tx, uint64_t object);
662 void dmu_tx_hold_spill(dmu_tx_t *tx, uint64_t object);
663 void dmu_tx_hold_sa(dmu_tx_t *tx, struct sa_handle *hdl, boolean_t may_grow);
664 void dmu_tx_hold_sa_create(dmu_tx_t *tx, int total_size);
665 void dmu_tx_abort(dmu_tx_t *tx);
666 int dmu_tx_assign(dmu_tx_t *tx, enum txg_how txg_how);
667 void dmu_tx_wait(dmu_tx_t *tx);
668 void dmu_tx_commit(dmu_tx_t *tx);
669 void dmu_tx_mark_netfree(dmu_tx_t *tx);

671 /*
672  * To register a commit callback, dmu_tx_callback_register() must be called.
673  *
674  * dcb_data is a pointer to caller private data that is passed on as a
675  * callback parameter. The caller is responsible for properly allocating and
676  * freeing it.
677  *
678  * When registering a callback, the transaction must be already created, but
679  * it cannot be committed or aborted. It can be assigned to a txg or not.
```



```
680  *
681  * The callback will be called after the transaction has been safely written
682  * to stable storage and will also be called if the dmu_tx is aborted.
683  * If there is any error which prevents the transaction from being committed to
684  * disk, the callback will be called with a value of error != 0.
685  */
686 typedef void dmu_tx_callback_func_t(void *dcb_data, int error);

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

691 /*
692  * Free up the data blocks for a defined range of a file.  If size is
693  * -1, the range from offset to end-of-file is freed.
694  */
695 int dmu_free_range(objset_t *os, uint64_t object, uint64_t offset,
696         uint64_t size, dmu_tx_t *tx);
697 int dmu_free_long_range(objset_t *os, uint64_t object, uint64_t offset,
698         uint64_t size);
699 int dmu_free_long_object(objset_t *os, uint64_t object);

701 /*
702  * Convenience functions.
703  *
704  * Canfail routines will return 0 on success, or an errno if there is a
705  * nonrecoverable I/O error.
706  */
707 #define DMU_READ_PREFETCH       0 /* prefetch */
708 #define DMU_READ_NO_PREFETCH    1 /* don't prefetch */
709 int dmu_read(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
710         void *buf, uint32_t flags);
711 void dmu_write(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
712         const void *buf, dmu_tx_t *tx);
713 void dmu_prealloc(objset_t *os, uint64_t object, uint64_t offset, uint64_t size,
714         dmu_tx_t *tx);
715 int dmu_read_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size);
716 int dmu_read_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size);
717 int dmu_write_uio(objset_t *os, uint64_t object, struct uio *uio, uint64_t size,
718     dmu_tx_t *tx);
719 int dmu_write_uio_dbuf(dmu_buf_t *zdb, struct uio *uio, uint64_t size,
720     dmu_tx_t *tx);
721 int dmu_write_pages(objset_t *os, uint64_t object, uint64_t offset,
722     uint64_t size, struct page *pp, dmu_tx_t *tx);
723 struct arc_buf *dmu_request_arcbuf(dmu_buf_t *handle, int size);
724 void dmu_return_arcbuf(struct arc_buf *buf);
725 void dmu_assign_arcbuf(dmu_buf_t *handle, uint64_t offset, struct arc_buf *buf,
726     dmu_tx_t *tx);
727 int dmu_xuio_init(struct xuio *uio, int niov);
728 void dmu_xuio_fini(struct xuio *uio);
729 int dmu_xuio_add(struct xuio *uio, struct arc_buf *abuf, offset_t off,
730     size_t n);
731 int dmu_xuio_cnt(struct xuio *uio);
732 struct arc_buf *dmu_xuio_arcbuf(struct xuio *uio, int i);
733 void dmu_xuio_clear(struct xuio *uio, int i);
734 void xuio_stat_wbuf_copied();
735 void xuio_stat_wbuf_nocopy();

737 extern int zfs_prefetch_disable;
738 extern int zfs_max_recordsize;

740 /*
741  * Asynchronously try to read in the data.
742  */
743 void dmu_prefetch(objset_t *os, uint64_t object, uint64_t offset,
744     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_

 128 #define DMU_META_OBJSET         0
 129 #define DMU_META_DNODE_OBJECT   0
 130 #define DMU_OBJECT_IS_SPECIAL(obj) ((int64_t)(obj) <= 0)
 131 #define DMU_META_DNODE(os)      ((os)->os_meta_dnode.dnh_dnode)
 132 #define DMU_USERUSED_DNODE(os)  ((os)->os_userused_dnode.dnh_dnode)
 133 #define DMU_GROUPUSED_DNODE(os) ((os)->os_groupused_dnode.dnh_dnode)

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

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

 141 /* called from zpl */
 142 int dmu_objset_hold(const char *name, void *tag, objset_t **osp);
 143 int dmu_objset_own(const char *name, dmu_objset_type_t type,
 144     boolean_t readonly, void *tag, objset_t **osp);
 145 int dmu_objset_own_obj(struct dsl_pool *dp, uint64_t obj,
 146     dmu_objset_type_t type, boolean_t readonly, void *tag, objset_t **osp);
 147 #endif /* ! codereview */
 148 void dmu_objset_refresh_ownership(objset_t *os, void *tag);
 149 void dmu_objset_rele(objset_t *os, void *tag);
 150 void dmu_objset_disown(objset_t *os, void *tag);
 151 int dmu_objset_from_ds(struct dsl_dataset *ds, objset_t **osp);

 153 void dmu_objset_stats(objset_t *os, nvlist_t *nv);
 154 void dmu_objset_fast_stat(objset_t *os, dmu_objset_stats_t *stat);
 155 void dmu_objset_space(objset_t *os, uint64_t *refdbytesp, uint64_t *availbytesp,
 156     uint64_t *usedobjsp, uint64_t *availobjsp);
 157 uint64_t dmu_objset_fsid_guid(objset_t *os);
 158 int dmu_objset_find_dp(struct dsl_pool *dp, uint64_t ddobj,
 159     int func(struct dsl_pool *, struct dsl_dataset *, void *),
 160     void *arg, int flags);
 161 int dmu_objset_prefetch(const char *name, void *arg);
 162 void dmu_objset_evict_dbufs(objset_t *os);
 163 timestruc_t dmu_objset_snap_cmtime(objset_t *os);

 165 /* called from dsl */
 166 void dmu_objset_sync(objset_t *os, zio_t *zio, dmu_tx_t *tx);
 167 boolean_t dmu_objset_is_dirty(objset_t *os, uint64_t txg);
 168 objset_t *dmu_objset_create_impl(spa_t *spa, struct dsl_dataset *ds,
 169     blkptr_t *bp, dmu_objset_type_t type, dmu_tx_t *tx);
 170 int dmu_objset_open_impl(spa_t *spa, struct dsl_dataset *ds, blkptr_t *bp,
 171     objset_t **osp);
 172 void dmu_objset_evict(objset_t *os);
 173 void dmu_objset_do_userquota_updates(objset_t *os, dmu_tx_t *tx);
 174 void dmu_objset_userquota_get_ids(dnode_t *dn, boolean_t before, dmu_tx_t *tx);
 175 boolean_t dmu_objset_userused_enabled(objset_t *os);
 176 int dmu_objset_userspace_upgrade(objset_t *os);
```



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

 180 void dmu_objset_evict_done(objset_t *os);

 182 void dmu_objset_init(void);
 183 void dmu_objset_fini(void);

 185 #ifdef  __cplusplus
 186 }
 187 #endif

 189 #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_

 135 int dsl_pool_init(spa_t *spa, uint64_t txg, dsl_pool_t **dpp);
 136 int dsl_pool_open(dsl_pool_t *dp);
 137 void dsl_pool_close(dsl_pool_t *dp);
 138 dsl_pool_t *dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg);
 139 void dsl_pool_sync(dsl_pool_t *dp, uint64_t txg);
 140 void dsl_pool_sync_done(dsl_pool_t *dp, uint64_t txg);
 141 int dsl_pool_sync_context(dsl_pool_t *dp);
 142 uint64_t dsl_pool_adjustedsize(dsl_pool_t *dp, boolean_t netfree);
 143 uint64_t dsl_pool_adjustedfree(dsl_pool_t *dp, boolean_t netfree);
 144 void dsl_pool_dirty_space(dsl_pool_t *dp, int64_t space, dmu_tx_t *tx);
 145 void dsl_pool_undirty_space(dsl_pool_t *dp, int64_t space, uint64_t txg);
 146 void dsl_free(dsl_pool_t *dp, uint64_t txg, const blkptr_t *bpp);
 147 void dsl_free_sync(zio_t *pio, dsl_pool_t *dp, uint64_t txg,
 148     const blkptr_t *bpp);
 149 void dsl_pool_create_origin(dsl_pool_t *dp, dmu_tx_t *tx);
 150 void dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx);
 151 void dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx);
 152 void dsl_pool_mos_diduse_space(dsl_pool_t *dp,
 153     int64_t used, int64_t comp, int64_t uncomp);
 154 void dsl_pool_config_enter(dsl_pool_t *dp, void *tag);
 155 void dsl_pool_config_exit(dsl_pool_t *dp, void *tag);
 156 boolean_t dsl_pool_config_held(dsl_pool_t *dp);
 157 boolean_t dsl_pool_config_held_writer(dsl_pool_t *dp);
 158 #endif /* ! codereview */
 159 boolean_t dsl_pool_need_dirty_delay(dsl_pool_t *dp);

 161 taskq_t *dsl_pool_vnrele_taskq(dsl_pool_t *dp);

 163 int dsl_pool_user_hold(dsl_pool_t *dp, uint64_t dsobj,
 164     const char *tag, uint64_t now, dmu_tx_t *tx);
 165 int dsl_pool_user_release(dsl_pool_t *dp, uint64_t dsobj,
 166     const char *tag, dmu_tx_t *tx);
 167 void dsl_pool_clean_tmp_userrefs(dsl_pool_t *dp);
 168 int dsl_pool_open_special_dir(dsl_pool_t *dp, const char *name, dsl_dir_t **);
 169 int dsl_pool_hold(const char *name, void *tag, dsl_pool_t **dp);
 170 void dsl_pool_rele(dsl_pool_t *dp, void *tag);

 172 #ifdef  __cplusplus
 173 }
 174 #endif

 176 #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_
```

```
 48 extern boolean_t zfs_nocacheflush;

 50 extern int vdev_open(vdev_t *);
 51 extern void vdev_open_children(vdev_t *);
 52 extern boolean_t vdev_uses_zvols(vdev_t *);
 53 extern int vdev_validate(vdev_t *, boolean_t);
 54 extern void vdev_close(vdev_t *);
 55 extern int vdev_create(vdev_t *, uint64_t txg, boolean_t isreplace);
 56 extern void vdev_reopen(vdev_t *);
 57 extern int vdev_validate_aux(vdev_t *vd);
 58 extern zio_t *vdev_probe(vdev_t *vd, zio_t *pio);

 60 extern boolean_t vdev_is_bootable(vdev_t *vd);
 61 extern vdev_t *vdev_lookup_top(spa_t *spa, uint64_t vdev);
 62 extern vdev_t *vdev_lookup_by_guid(vdev_t *vd, uint64_t guid);
 63 extern int vdev_count_leaves(spa_t *spa);
 64 #endif /* ! codereview */
 65 extern void vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t d,
 66     uint64_t txg, uint64_t size);
 67 extern boolean_t vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t d,
 68     uint64_t txg, uint64_t size);
 69 extern boolean_t vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t d);
 70 extern void vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg,
 71     int scrub_done);
 72 extern boolean_t vdev_dtl_required(vdev_t *vd);
 73 extern boolean_t vdev_resilver_needed(vdev_t *vd,
 74     uint64_t *minp, uint64_t *maxp);

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

 79 extern int vdev_metaslab_init(vdev_t *vd, uint64_t txg);
 80 extern void vdev_metaslab_fini(vdev_t *vd);
 81 extern void vdev_metaslab_set_size(vdev_t *);
 82 extern void vdev_expand(vdev_t *vd, uint64_t txg);
 83 extern void vdev_split(vdev_t *vd);
 84 extern void vdev_deadman(vdev_t *vd);


 87 extern void vdev_get_stats(vdev_t *vd, vdev_stat_t *vs);
 88 extern void vdev_clear_stats(vdev_t *vd);
 89 extern void vdev_stat_update(zio_t *zio, uint64_t psize);
 90 extern void vdev_scan_stat_init(vdev_t *vd);
 91 extern void vdev_propagate_state(vdev_t *vd);
 92 extern void vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state,
 93     vdev_aux_t aux);

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



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

100 extern int vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux);
101 extern int vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux);
102 extern int vdev_online(spa_t *spa, uint64_t guid, uint64_t flags,
103     vdev_state_t *);
104 extern int vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags);
105 extern void vdev_clear(spa_t *spa, vdev_t *vd);

107 extern boolean_t vdev_is_dead(vdev_t *vd);
108 extern boolean_t vdev_readable(vdev_t *vd);
109 extern boolean_t vdev_writeable(vdev_t *vd);
110 extern boolean_t vdev_allocatable(vdev_t *vd);
111 extern boolean_t vdev_accessible(vdev_t *vd, zio_t *zio);

113 extern void vdev_cache_init(vdev_t *vd);
114 extern void vdev_cache_fini(vdev_t *vd);
115 extern boolean_t vdev_cache_read(zio_t *zio);
116 extern void vdev_cache_write(zio_t *zio);
117 extern void vdev_cache_purge(vdev_t *vd);

119 extern void vdev_queue_init(vdev_t *vd);
120 extern void vdev_queue_fini(vdev_t *vd);
121 extern zio_t *vdev_queue_io(zio_t *zio);
122 extern void vdev_queue_io_done(zio_t *zio);

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

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

132 typedef enum vdev_config_flag {
133         VDEV_CONFIG_SPARE = 1 << 0,
134         VDEV_CONFIG_L2CACHE = 1 << 1,
135         VDEV_CONFIG_REMOVING = 1 << 2
136 } vdev_config_flag_t;

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

142 /*
143  * Label routines
144  */
145 struct uberblock;
146 extern uint64_t vdev_label_offset(uint64_t psize, int l, uint64_t offset);
147 extern int vdev_label_number(uint64_t psise, uint64_t offset);
148 extern nvlist_t *vdev_label_read_config(vdev_t *vd, uint64_t txg);
149 extern void vdev_uberblock_load(vdev_t *, struct uberblock *, nvlist_t **);

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

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

162 #ifdef  __cplusplus
```

```
163 }
164 #endif

166 #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>
**********************************************************
```

```
   1 /*
   2  * CDDL HEADER START
   3  *
   4  * The contents of this file are subject to the terms of the
   5  * Common Development and Distribution License (the "License").
   6  * You may not use this file except in compliance with the License.
   7  *
   8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
   9  * or http://www.opensolaris.org/os/licensing.
  10  * See the License for the specific language governing permissions
  11  * and limitations under the License.
  12  *
  13  * When distributing Covered Code, include this CDDL HEADER in each
  14  * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
  15  * If applicable, add the following below this CDDL HEADER, with the
  16  * fields enclosed by brackets "[]" replaced with your own identifying
  17  * information: Portions Copyright [yyyy] [name of copyright owner]
  18  *
  19  * CDDL HEADER END
  20  */
  21 /*
  22  * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
  23  * Copyright (c) 2012 by Delphix. All rights reserved.
  24  */

  26 /* Portions Copyright 2010 Robert Milkowski */

  28 #ifndef _SYS_ZIL_H
  29 #define _SYS_ZIL_H

  31 #include <sys/types.h>
  32 #include <sys/spa.h>
  33 #include <sys/zio.h>
  34 #include <sys/dmu.h>

  36 #ifdef  __cplusplus
  37 extern "C" {
  38 #endif

  40 struct dsl_pool;
  41 struct dsl_dataset;

  43 #endif /* ! codereview */
  44 /*
  45  * Intent log format:
  46  *
  47  * Each objset has its own intent log.  The log header (zil_header_t)
  48  * for objset N's intent log is kept in the Nth object of the SPA's
  49  * intent_log objset.  The log header points to a chain of log blocks,
  50  * each of which contains log records (i.e., transactions) followed by
  51  * a log block trailer (zil_trailer_t).  The format of a log record
```



```
  52  * depends on the record (or transaction) type, but all records begin
  53  * with a common structure that defines the type, length, and txg.
  54  */

  56 /*
  57  * Intent log header - this on disk structure holds fields to manage
  58  * the log.  All fields are 64 bit to easily handle cross architectures.
  59  */
  60 typedef struct zil_header {
  61         uint64_t zh_claim_txg;  /* txg in which log blocks were claimed */
  62         uint64_t zh_replay_seq; /* highest replayed sequence number */
  63         blkptr_t zh_log;        /* log chain */
  64         uint64_t zh_claim_blk_seq; /* highest claimed block sequence number */
  65         uint64_t zh_flags;      /* header flags */
  66         uint64_t zh_claim_lr_seq; /* highest claimed lr sequence number */
  67         uint64_t zh_pad[3];
  68 } zil_header_t;

  70 /*
  71  * zh_flags bit settings
  72  */
  73 #define ZIL_REPLAY_NEEDED       0x1     /* replay needed - internal only */
  74 #define ZIL_CLAIM_LR_SEQ_VALID  0x2     /* zh_claim_lr_seq field is valid */

  76 /*
  77  * Log block chaining.
  78  *
  79  * Log blocks are chained together. Originally they were chained at the
  80  * end of the block. For performance reasons the chain was moved to the
  81  * beginning of the block which allows writes for only the data being used.
  82  * The older position is supported for backwards compatability.
  83  *
  84  * The zio_eck_t contains a zec_cksum which for the intent log is
  85  * the sequence number of this log block. A seq of 0 is invalid.
  86  * The zec_cksum is checked by the SPA against the sequence
  87  * number passed in the blk_cksum field of the blkptr_t
  88  */
  89 typedef struct zil_chain {
  90         uint64_t zc_pad;
  91         blkptr_t zc_next_blk;   /* next block in chain */
  92         uint64_t zc_nused;      /* bytes in log block used */
  93         zio_eck_t zc_eck;       /* block trailer */
  94 } zil_chain_t;

  96 #define ZIL_MIN_BLKSZ   4096ULL

  98 /*
  99  * The words of a log block checksum.
 100  */
 101 #define ZIL_ZC_GUID_0   0
 102 #define ZIL_ZC_GUID_1   1
 103 #define ZIL_ZC_OBJSET   2
 104 #define ZIL_ZC_SEQ      3

 106 typedef enum zil_create {
 107         Z_FILE,
 108         Z_DIR,
 109         Z_XATTRDIR,
 110 } zil_create_t;

 112 /*
 113  * size of xvattr log section.
 114  * its composed of lr_attr_t + xvattr bitmap + 2 64 bit timestamps
 115  * for create time and a single 64 bit integer for all of the attributes,
 116  * and 4 64 bit integers (32 bytes) for the scanstamp.
 117  *
```

```
 */

#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);
```

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

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

388 extern void     zil_init(void);
389 extern void     zil_fini(void);

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

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

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

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

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

410 extern int      zil_vdev_offline(const char *osname, void *txarg);
411 extern int      zil_claim(struct dsl_pool *dp,
412     struct dsl_dataset *ds, void *txarg);
413 extern int      zil_check_log_chain(struct dsl_pool *dp,
414     struct dsl_dataset *ds, void *tx);
 40 extern int      zil_claim(const char *osname, void *txarg);
 41 extern int      zil_check_log_chain(const char *osname, void *txarg);
415 extern void     zil_sync(zilog_t *zilog, dmu_tx_t *tx);
416 extern void     zil_clean(zilog_t *zilog, uint64_t synced_txg);

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

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

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

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

428 extern int zil_replay_disable;

430 #ifdef  __cplusplus
431 }
```

*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_
```

```
181 static int
182 vdev_count_leaves_impl(vdev_t *vd)
183 {
184         int n = 0;
185
186         if (vd->vdev_ops->vdev_op_leaf)
187                 return (1);
188
189         for (int c = 0; c < vd->vdev_children; c++)
190                 n += vdev_count_leaves_impl(vd->vdev_child[c]);
191
192         return (n);
193 }
194
195 int
196 vdev_count_leaves(spa_t *spa)
197 {
198         return (vdev_count_leaves_impl(spa->spa_root_vdev));
199 }
200
201 #endif /* ! codereview */
202 void
203 vdev_add_child(vdev_t *pvd, vdev_t *cvd)
204 {
205         size_t oldsize, newsize;
206         uint64_t id = cvd->vdev_id;
207         vdev_t **newchild;
208
209         ASSERT(spa_config_held(cvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
210         ASSERT(cvd->vdev_parent == NULL);
211
212         cvd->vdev_parent = pvd;
213
214         if (pvd == NULL)
215                 return;
216
217         ASSERT(id >= pvd->vdev_children || pvd->vdev_child[id] == NULL);
218
219         oldsize = pvd->vdev_children * sizeof (vdev_t *);
220         pvd->vdev_children = MAX(pvd->vdev_children, id + 1);
221         newsize = pvd->vdev_children * sizeof (vdev_t *);
222
223         newchild = kmem_zalloc(newsize, KM_SLEEP);
224         if (pvd->vdev_child != NULL) {
225                 bcopy(pvd->vdev_child, newchild, oldsize);
226                 kmem_free(pvd->vdev_child, oldsize);
227         }
228
229         pvd->vdev_child = newchild;
```



```
230         pvd->vdev_child[id] = cvd;
231
232         cvd->vdev_top = (pvd->vdev_top ? pvd->vdev_top: cvd);
233         ASSERT(cvd->vdev_top->vdev_parent->vdev_parent == NULL);
234
235         /*
236          * Walk up all ancestors to update guid sum.
237          */
238         for (; pvd != NULL; pvd = pvd->vdev_parent)
239                 pvd->vdev_guid_sum += cvd->vdev_guid_sum;
240 }
241
242 void
243 vdev_remove_child(vdev_t *pvd, vdev_t *cvd)
244 {
245         int c;
246         uint_t id = cvd->vdev_id;
247
248         ASSERT(cvd->vdev_parent == pvd);
249
250         if (pvd == NULL)
251                 return;
252
253         ASSERT(id < pvd->vdev_children);
254         ASSERT(pvd->vdev_child[id] == cvd);
255
256         pvd->vdev_child[id] = NULL;
257         cvd->vdev_parent = NULL;
258
259         for (c = 0; c < pvd->vdev_children; c++)
260                 if (pvd->vdev_child[c])
261                         break;
262
263         if (c == pvd->vdev_children) {
264                 kmem_free(pvd->vdev_child, c * sizeof (vdev_t *));
265                 pvd->vdev_child = NULL;
266                 pvd->vdev_children = 0;
267         }
268
269         /*
270          * Walk up all ancestors to update guid sum.
271          */
272         for (; pvd != NULL; pvd = pvd->vdev_parent)
273                 pvd->vdev_guid_sum -= cvd->vdev_guid_sum;
274 }
275
276 /*
277  * Remove any holes in the child array.
278  */
279 void
280 vdev_compact_children(vdev_t *pvd)
281 {
282         vdev_t **newchild, *cvd;
283         int oldc = pvd->vdev_children;
284         int newc;
285
286         ASSERT(spa_config_held(pvd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);
287
288         for (int c = newc = 0; c < oldc; c++)
289                 if (pvd->vdev_child[c])
290                         newc++;
291
292         newchild = kmem_alloc(newc * sizeof (vdev_t *), KM_SLEEP);
293
294         for (int c = newc = 0; c < oldc; c++) {
295                 if ((cvd = pvd->vdev_child[c]) != NULL) {
```

```
296                         newchild[newc] = cvd;
297                         cvd->vdev_id = newc++;
298                 }
299         }

301         kmem_free(pvd->vdev_child, oldc * sizeof (vdev_t *));
302         pvd->vdev_child = newchild;
303         pvd->vdev_children = newc;
304 }

306 /*
307  * Allocate and minimally initialize a vdev_t.
308  */
309 vdev_t *
310 vdev_alloc_common(spa_t *spa, uint_t id, uint64_t guid, vdev_ops_t *ops)
311 {
312         vdev_t *vd;

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

316         if (spa->spa_root_vdev == NULL) {
317                 ASSERT(ops == &vdev_root_ops);
318                 spa->spa_root_vdev = vd;
319                 spa->spa_load_guid = spa_generate_guid(NULL);
320         }

322         if (guid == 0 && ops != &vdev_hole_ops) {
323                 if (spa->spa_root_vdev == vd) {
324                         /*
325                          * The root vdev's guid will also be the pool guid,
326                          * which must be unique among all pools.
327                          */
328                         guid = spa_generate_guid(NULL);
329                 } else {
330                         /*
331                          * Any other vdev's guid must be unique within the pool.
332                          */
333                         guid = spa_generate_guid(spa);
334                 }
335                 ASSERT(!spa_guid_exists(spa_guid(spa), guid));
336         }

338         vd->vdev_spa = spa;
339         vd->vdev_id = id;
340         vd->vdev_guid = guid;
341         vd->vdev_guid_sum = guid;
342         vd->vdev_ops = ops;
343         vd->vdev_state = VDEV_STATE_CLOSED;
344         vd->vdev_ishole = (ops == &vdev_hole_ops);

346         mutex_init(&vd->vdev_dtl_lock, NULL, MUTEX_DEFAULT, NULL);
347         mutex_init(&vd->vdev_stat_lock, NULL, MUTEX_DEFAULT, NULL);
348         mutex_init(&vd->vdev_probe_lock, NULL, MUTEX_DEFAULT, NULL);
349         for (int t = 0; t < DTL_TYPES; t++) {
350                 vd->vdev_dtl[t] = range_tree_create(NULL, NULL,
351                     &vd->vdev_dtl_lock);
352         }
353         txg_list_create(&vd->vdev_ms_list,
354             offsetof(struct metaslab, ms_txg_node));
355         txg_list_create(&vd->vdev_dtl_list,
356             offsetof(struct vdev, vdev_dtl_node));
357         vd->vdev_stat.vs_timestamp = gethrtime();
358         vdev_queue_init(vd);
359         vdev_cache_init(vd);

361         return (vd);
```



```
362 }

364 /*
365  * Allocate a new vdev.  The 'alloctype' is used to control whether we are
366  * creating a new vdev or loading an existing one - the behavior is slightly
367  * different for each case.
368  */
369 int
370 vdev_alloc(spa_t *spa, vdev_t **vdp, nvlist_t *nv, vdev_t *parent, uint_t id,
371     int alloctype)
372 {
373         vdev_ops_t *ops;
374         char *type;
375         uint64_t guid = 0, islog, nparity;
376         vdev_t *vd;

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

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

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

386         /*
387          * If this is a load, get the vdev guid from the nvlist.
388          * Otherwise, vdev_alloc_common() will generate one for us.
389          */
390         if (alloctype == VDEV_ALLOC_LOAD) {
391                 uint64_t label_id;

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

397                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
398                         return (SET_ERROR(EINVAL));
399         } else if (alloctype == VDEV_ALLOC_SPARE) {
400                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
401                         return (SET_ERROR(EINVAL));
402         } else if (alloctype == VDEV_ALLOC_L2CACHE) {
403                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
404                         return (SET_ERROR(EINVAL));
405         } else if (alloctype == VDEV_ALLOC_ROOTPOOL) {
406                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_GUID, &guid) != 0)
407                         return (SET_ERROR(EINVAL));
408         }

410         /*
411          * The first allocated vdev must be of type 'root'.
412          */
413         if (ops != &vdev_root_ops && spa->spa_root_vdev == NULL)
414                 return (SET_ERROR(EINVAL));

416         /*
417          * Determine whether we're a log vdev.
418          */
419         islog = 0;
420         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_LOG, &islog);
421         if (islog && spa_version(spa) < SPA_VERSION_SLOGS)
422                 return (SET_ERROR(ENOTSUP));

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

427         /*
```

```
428          * Set the nparity property for RAID-Z vdevs.
429          */
430         nparity = -1ULL;
431         if (ops == &vdev_raidz_ops) {
432                 if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NPARITY,
433                     &nparity) == 0) {
434                         if (nparity == 0 || nparity > VDEV_RAIDZ_MAXPARITY)
435                                 return (SET_ERROR(EINVAL));
436                         /*
437                          * Previous versions could only support 1 or 2 parity
438                          * device.
439                          */
440                         if (nparity > 1 &&
441                             spa_version(spa) < SPA_VERSION_RAIDZ2)
442                                 return (SET_ERROR(ENOTSUP));
443                         if (nparity > 2 &&
444                             spa_version(spa) < SPA_VERSION_RAIDZ3)
445                                 return (SET_ERROR(ENOTSUP));
446                 } else {
447                         /*
448                          * We require the parity to be specified for SPAs that
449                          * support multiple parity levels.
450                          */
451                         if (spa_version(spa) >= SPA_VERSION_RAIDZ2)
452                                 return (SET_ERROR(EINVAL));
453                         /*
454                          * Otherwise, we default to 1 parity device for RAID-Z.
455                          */
456                         nparity = 1;
457                 }
458         } else {
459                 nparity = 0;
460         }
461         ASSERT(nparity != -1ULL);

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

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

468         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PATH, &vd->vdev_path) == 0)
469                 vd->vdev_path = spa_strdup(vd->vdev_path);
470         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_DEVID, &vd->vdev_devid) == 0)
471                 vd->vdev_devid = spa_strdup(vd->vdev_devid);
472         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_PHYS_PATH,
473             &vd->vdev_physpath) == 0)
474                 vd->vdev_physpath = spa_strdup(vd->vdev_physpath);
475         if (nvlist_lookup_string(nv, ZPOOL_CONFIG_FRU, &vd->vdev_fru) == 0)
476                 vd->vdev_fru = spa_strdup(vd->vdev_fru);

478         /*
479          * Set the whole_disk property.  If it's not specified, leave the value
480          * as -1.
481          */
482         if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_WHOLE_DISK,
483             &vd->vdev_wholedisk) != 0)
484                 vd->vdev_wholedisk = -1ULL;

486         /*
487          * Look for the 'not present' flag.  This will only be set if the device
488          * was not present at the time of import.
489          */
490         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_NOT_PRESENT,
491             &vd->vdev_not_present);

493         /*
```



```
494          * Get the alignment requirement.
495          */
496         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASHIFT, &vd->vdev_ashift);

498         /*
499          * Retrieve the vdev creation time.
500          */
501         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_CREATE_TXG,
502             &vd->vdev_crtxg);

504         /*
505          * If we're a top-level vdev, try to load the allocation parameters.
506          */
507         if (parent && !parent->vdev_parent &&
508             (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_SPLIT)) {
509                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_ARRAY,
510                     &vd->vdev_ms_array);
511                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_METASLAB_SHIFT,
512                     &vd->vdev_ms_shift);
513                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_ASIZE,
514                     &vd->vdev_asize);
515                 (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVING,
516                     &vd->vdev_removing);
517         }

519         if (parent && !parent->vdev_parent && alloctype != VDEV_ALLOC_ATTACH) {
520                 ASSERT(alloctype == VDEV_ALLOC_LOAD ||
521                     alloctype == VDEV_ALLOC_ADD ||
522                     alloctype == VDEV_ALLOC_SPLIT ||
523                     alloctype == VDEV_ALLOC_ROOTPOOL);
524                 vd->vdev_mg = metaslab_group_create(islog ?
525                     spa_log_class(spa) : spa_normal_class(spa), vd);
526         }

528         /*
529          * If we're a leaf vdev, try to load the DTL object and other state.
530          */
531         if (vd->vdev_ops->vdev_op_leaf &&
532             (alloctype == VDEV_ALLOC_LOAD || alloctype == VDEV_ALLOC_L2CACHE ||
533             alloctype == VDEV_ALLOC_ROOTPOOL)) {
534                 if (alloctype == VDEV_ALLOC_LOAD) {
535                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DTL,
536                             &vd->vdev_dtl_object);
537                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_UNSPARE,
538                             &vd->vdev_unspare);
539                 }

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

544                         if (nvlist_lookup_uint64(nv, ZPOOL_CONFIG_IS_SPARE,
545                             &spare) == 0 && spare)
546                                 spa_spare_add(vd);
547                 }

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

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

555                 /*
556                  * When importing a pool, we want to ignore the persistent fault
557                  * state, as the diagnosis made on another system may not be
558                  * valid in the current context.  Local vdevs will
559                  * remain in the faulted state.
```

```
560                  */
561                 if (spa_load_state(spa) == SPA_LOAD_OPEN) {
562                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_FAULTED,
563                             &vd->vdev_faulted);
564                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_DEGRADED,
565                             &vd->vdev_degraded);
566                         (void) nvlist_lookup_uint64(nv, ZPOOL_CONFIG_REMOVED,
567                             &vd->vdev_removed);
568
569                         if (vd->vdev_faulted || vd->vdev_degraded) {
570                                 char *aux;
571
572                                 vd->vdev_label_aux =
573                                     VDEV_AUX_ERR_EXCEEDED;
574                                 if (nvlist_lookup_string(nv,
575                                     ZPOOL_CONFIG_AUX_STATE, &aux) == 0 &&
576                                     strcmp(aux, "external") == 0)
577                                         vd->vdev_label_aux = VDEV_AUX_EXTERNAL;
578                         }
579                 }
580         }
581
582         /*
583          * Add ourselves to the parent's list of children.
584          */
585         vdev_add_child(parent, vd);
586
587         *vdp = vd;
588
589         return (0);
590 }
591
592 void
593 vdev_free(vdev_t *vd)
594 {
595         spa_t *spa = vd->vdev_spa;
596
597         /*
598          * vdev_free() implies closing the vdev first.  This is simpler than
599          * trying to ensure complicated semantics for all callers.
600          */
601         vdev_close(vd);
602
603         ASSERT(!list_link_active(&vd->vdev_config_dirty_node));
604         ASSERT(!list_link_active(&vd->vdev_state_dirty_node));
605
606         /*
607          * Free all children.
608          */
609         for (int c = 0; c < vd->vdev_children; c++)
610                 vdev_free(vd->vdev_child[c]);
611
612         ASSERT(vd->vdev_child == NULL);
613         ASSERT(vd->vdev_guid_sum == vd->vdev_guid);
614
615         /*
616          * Discard allocation state.
617          */
618         if (vd->vdev_mg != NULL) {
619                 vdev_metaslab_fini(vd);
620                 metaslab_group_destroy(vd->vdev_mg);
621         }
622
623         ASSERT0(vd->vdev_stat.vs_space);
624         ASSERT0(vd->vdev_stat.vs_dspace);
625         ASSERT0(vd->vdev_stat.vs_alloc);
```



```
626
627         /*
628          * Remove this vdev from its parent's child list.
629          */
630         vdev_remove_child(vd->vdev_parent, vd);
631
632         ASSERT(vd->vdev_parent == NULL);
633
634         /*
635          * Clean up vdev structure.
636          */
637         vdev_queue_fini(vd);
638         vdev_cache_fini(vd);
639
640         if (vd->vdev_path)
641                 spa_strfree(vd->vdev_path);
642         if (vd->vdev_devid)
643                 spa_strfree(vd->vdev_devid);
644         if (vd->vdev_physpath)
645                 spa_strfree(vd->vdev_physpath);
646         if (vd->vdev_fru)
647                 spa_strfree(vd->vdev_fru);
648
649         if (vd->vdev_isspare)
650                 spa_spare_remove(vd);
651         if (vd->vdev_isl2cache)
652                 spa_l2cache_remove(vd);
653
654         txg_list_destroy(&vd->vdev_ms_list);
655         txg_list_destroy(&vd->vdev_dtl_list);
656
657         mutex_enter(&vd->vdev_dtl_lock);
658         space_map_close(vd->vdev_dtl_sm);
659         for (int t = 0; t < DTL_TYPES; t++) {
660                 range_tree_vacate(vd->vdev_dtl[t], NULL, NULL);
661                 range_tree_destroy(vd->vdev_dtl[t]);
662         }
663         mutex_exit(&vd->vdev_dtl_lock);
664
665         mutex_destroy(&vd->vdev_dtl_lock);
666         mutex_destroy(&vd->vdev_stat_lock);
667         mutex_destroy(&vd->vdev_probe_lock);
668
669         if (vd == spa->spa_root_vdev)
670                 spa->spa_root_vdev = NULL;
671
672         kmem_free(vd, sizeof (vdev_t));
673 }
674
675 /*
676  * Transfer top-level vdev state from svd to tvd.
677  */
678 static void
679 vdev_top_transfer(vdev_t *svd, vdev_t *tvd)
680 {
681         spa_t *spa = svd->vdev_spa;
682         metaslab_t *msp;
683         vdev_t *vd;
684         int t;
685
686         ASSERT(tvd == tvd->vdev_top);
687
688         tvd->vdev_ms_array = svd->vdev_ms_array;
689         tvd->vdev_ms_shift = svd->vdev_ms_shift;
690         tvd->vdev_ms_count = svd->vdev_ms_count;
```

```
692         svd->vdev_ms_array = 0;
693         svd->vdev_ms_shift = 0;
694         svd->vdev_ms_count = 0;

696         if (tvd->vdev_mg)
697                 ASSERT3P(tvd->vdev_mg, ==, svd->vdev_mg);
698         tvd->vdev_mg = svd->vdev_mg;
699         tvd->vdev_ms = svd->vdev_ms;

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

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

707         tvd->vdev_stat.vs_alloc = svd->vdev_stat.vs_alloc;
708         tvd->vdev_stat.vs_space = svd->vdev_stat.vs_space;
709         tvd->vdev_stat.vs_dspace = svd->vdev_stat.vs_dspace;

711         svd->vdev_stat.vs_alloc = 0;
712         svd->vdev_stat.vs_space = 0;
713         svd->vdev_stat.vs_dspace = 0;

715         for (t = 0; t < TXG_SIZE; t++) {
716                 while ((msp = txg_list_remove(&svd->vdev_ms_list, t)) != NULL)
717                         (void) txg_list_add(&tvd->vdev_ms_list, msp, t);
718                 while ((vd = txg_list_remove(&svd->vdev_dtl_list, t)) != NULL)
719                         (void) txg_list_add(&tvd->vdev_dtl_list, vd, t);
720                 if (txg_list_remove_this(&spa->spa_vdev_txg_list, svd, t))
721                         (void) txg_list_add(&spa->spa_vdev_txg_list, tvd, t);
722         }

724         if (list_link_active(&svd->vdev_config_dirty_node)) {
725                 vdev_config_clean(svd);
726                 vdev_config_dirty(tvd);
727         }

729         if (list_link_active(&svd->vdev_state_dirty_node)) {
730                 vdev_state_clean(svd);
731                 vdev_state_dirty(tvd);
732         }

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

737         tvd->vdev_islog = svd->vdev_islog;
738         svd->vdev_islog = 0;
739 }

741 static void
742 vdev_top_update(vdev_t *tvd, vdev_t *vd)
743 {
744         if (vd == NULL)
745                 return;

747         vd->vdev_top = tvd;

749         for (int c = 0; c < vd->vdev_children; c++)
750                 vdev_top_update(tvd, vd->vdev_child[c]);
751 }

753 /*
754  * Add a mirror/replacing vdev above an existing vdev.
755  */
756 vdev_t *
757 vdev_add_parent(vdev_t *cvd, vdev_ops_t *ops)
```



```
758 {
759         spa_t *spa = cvd->vdev_spa;
760         vdev_t *pvd = cvd->vdev_parent;
761         vdev_t *mvd;

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

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

767         mvd->vdev_asize = cvd->vdev_asize;
768         mvd->vdev_min_asize = cvd->vdev_min_asize;
769         mvd->vdev_max_asize = cvd->vdev_max_asize;
770         mvd->vdev_ashift = cvd->vdev_ashift;
771         mvd->vdev_state = cvd->vdev_state;
772         mvd->vdev_crtxg = cvd->vdev_crtxg;

774         vdev_remove_child(pvd, cvd);
775         vdev_add_child(pvd, mvd);
776         cvd->vdev_id = mvd->vdev_children;
777         vdev_add_child(mvd, cvd);
778         vdev_top_update(cvd->vdev_top, cvd->vdev_top);

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

783         return (mvd);
784 }

786 /*
787  * Remove a 1-way mirror/replacing vdev from the tree.
788  */
789 void
790 vdev_remove_parent(vdev_t *cvd)
791 {
792         vdev_t *mvd = cvd->vdev_parent;
793         vdev_t *pvd = mvd->vdev_parent;

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

797         ASSERT(mvd->vdev_children == 1);
798         ASSERT(mvd->vdev_ops == &vdev_mirror_ops ||
799             mvd->vdev_ops == &vdev_replacing_ops ||
800             mvd->vdev_ops == &vdev_spare_ops);
801         cvd->vdev_ashift = mvd->vdev_ashift;

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

806         /*
807          * If cvd will replace mvd as a top-level vdev, preserve mvd's guid.
808          * Otherwise, we could have detached an offline device, and when we
809          * go to import the pool we'll think we have two top-level vdevs,
810          * instead of a different version of the same top-level vdev.
811          */
812         if (mvd->vdev_top == mvd) {
813                 uint64_t guid_delta = mvd->vdev_guid - cvd->vdev_guid;
814                 cvd->vdev_orig_guid = cvd->vdev_guid;
815                 cvd->vdev_guid += guid_delta;
816                 cvd->vdev_guid_sum += guid_delta;
817         }
818         cvd->vdev_id = mvd->vdev_id;
819         vdev_add_child(pvd, cvd);
820         vdev_top_update(cvd->vdev_top, cvd->vdev_top);

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

```
825         ASSERT(mvd->vdev_children == 0);
826         vdev_free(mvd);
827 }

829 int
830 vdev_metaslab_init(vdev_t *vd, uint64_t txg)
831 {
832         spa_t *spa = vd->vdev_spa;
833         objset_t *mos = spa->spa_meta_objset;
834         uint64_t m;
835         uint64_t oldc = vd->vdev_ms_count;
836         uint64_t newc = vd->vdev_asize >> vd->vdev_ms_shift;
837         metaslab_t **mspp;
838         int error;

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

842         /*
843          * This vdev is not being allocated from yet or is a hole.
844          */
845         if (vd->vdev_ms_shift == 0)
846                 return (0);

848         ASSERT(!vd->vdev_ishole);

850         /*
851          * Compute the raidz-deflation ratio.  Note, we hard-code
852          * in 128k (1 << 17) because it is the "typical" blocksize.
853          * Even though SPA_MAXBLOCKSIZE changed, this algorithm can not change,
854          * otherwise it would inconsistently account for existing bp's.
855          */
856         vd->vdev_deflate_ratio = (1 << 17) /
857             (vdev_psize_to_asize(vd, 1 << 17) >> SPA_MINBLOCKSHIFT);

859         ASSERT(oldc <= newc);

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

863         if (oldc != 0) {
864                 bcopy(vd->vdev_ms, mspp, oldc * sizeof (*mspp));
865                 kmem_free(vd->vdev_ms, oldc * sizeof (*mspp));
866         }

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

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

874                 if (txg == 0) {
875                         error = dmu_read(mos, vd->vdev_ms_array,
876                             m * sizeof (uint64_t), sizeof (uint64_t), &object,
877                             DMU_READ_PREFETCH);
878                         if (error)
879                                 return (error);
880                 }

882                 error = metaslab_init(vd->vdev_mg, m, object, txg,
883                     &(vd->vdev_ms[m]));
884                 if (error)
885                         return (error);
886         }

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



```
891         /*
892          * If the vdev is being removed we don't activate
893          * the metaslabs since we want to ensure that no new
894          * allocations are performed on this device.
895          */
896         if (oldc == 0 && !vd->vdev_removing)
897                 metaslab_group_activate(vd->vdev_mg);

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

902         return (0);
903 }

905 void
906 vdev_metaslab_fini(vdev_t *vd)
907 {
908         uint64_t m;
909         uint64_t count = vd->vdev_ms_count;

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

916                         if (msp != NULL)
917                                 metaslab_fini(msp);
918                 }
919                 kmem_free(vd->vdev_ms, count * sizeof (metaslab_t *));
920                 vd->vdev_ms = NULL;
921         }
922 }

924 typedef struct vdev_probe_stats {
925         boolean_t       vps_readable;
926         boolean_t       vps_writeable;
927         int             vps_flags;
928 } vdev_probe_stats_t;

930 static void
931 vdev_probe_done(zio_t *zio)
932 {
933         spa_t *spa = zio->io_spa;
934         vdev_t *vd = zio->io_vd;
935         vdev_probe_stats_t *vps = zio->io_private;

937         ASSERT(vd->vdev_probe_zio != NULL);

939         if (zio->io_type == ZIO_TYPE_READ) {
940                 if (zio->io_error == 0)
941                         vps->vps_readable = 1;
942                 if (zio->io_error == 0 && spa_writeable(spa)) {
943                         zio_nowait(zio_write_phys(vd->vdev_probe_zio, vd,
944                             zio->io_offset, zio->io_size, zio->io_data,
945                             ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
946                             ZIO_PRIORITY_SYNC_WRITE, vps->vps_flags, B_TRUE));
947                 } else {
948                         zio_buf_free(zio->io_data, zio->io_size);
949                 }
950         } else if (zio->io_type == ZIO_TYPE_WRITE) {
951                 if (zio->io_error == 0)
952                         vps->vps_writeable = 1;
953                 zio_buf_free(zio->io_data, zio->io_size);
954         } else if (zio->io_type == ZIO_TYPE_NULL) {
955                 zio_t *pio;
```

```
957                 vd->vdev_cant_read |= !vps->vps_readable;
958                 vd->vdev_cant_write |= !vps->vps_writeable;

960                 if (vdev_readable(vd) &&
961                     (vdev_writeable(vd) || !spa_writeable(spa))) {
962                         zio->io_error = 0;
963                 } else {
964                         ASSERT(zio->io_error != 0);
965                         zfs_ereport_post(FM_EREPORT_ZFS_PROBE_FAILURE,
966                             spa, vd, NULL, 0, 0);
967                         zio->io_error = SET_ERROR(ENXIO);
968                 }

970                 mutex_enter(&vd->vdev_probe_lock);
971                 ASSERT(vd->vdev_probe_zio == zio);
972                 vd->vdev_probe_zio = NULL;
973                 mutex_exit(&vd->vdev_probe_lock);

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

979                 kmem_free(vps, sizeof (*vps));
980         }
981 }

983 /*
984  * Determine whether this device is accessible.
985  *
986  * Read and write to several known locations: the pad regions of each
987  * vdev label but the first, which we leave alone in case it contains
988  * a VTOC.
989  */
990 zio_t *
991 vdev_probe(vdev_t *vd, zio_t *zio)
992 {
993         spa_t *spa = vd->vdev_spa;
994         vdev_probe_stats_t *vps = NULL;
995         zio_t *pio;

997         ASSERT(vd->vdev_ops->vdev_op_leaf);

999         /*
1000         * Don't probe the probe.
1001         */
1002        if (zio && (zio->io_flags & ZIO_FLAG_PROBE))
1003                return (NULL);

1005        /*
1006         * To prevent 'probe storms' when a device fails, we create
1007         * just one probe i/o at a time.  All zios that want to probe
1008         * this vdev will become parents of the probe io.
1009         */
1010        mutex_enter(&vd->vdev_probe_lock);

1012        if ((pio = vd->vdev_probe_zio) == NULL) {
1013                vps = kmem_zalloc(sizeof (*vps), KM_SLEEP);

1015                vps->vps_flags = ZIO_FLAG_CANFAIL | ZIO_FLAG_PROBE |
1016                    ZIO_FLAG_DONT_CACHE | ZIO_FLAG_DONT_AGGREGATE |
1017                    ZIO_FLAG_TRYHARD;

1019                if (spa_config_held(spa, SCL_ZIO, RW_WRITER)) {
1020                        /*
1021                         * vdev_cant_read and vdev_cant_write can only
```



```
1022                         * transition from TRUE to FALSE when we have the
1023                         * SCL_ZIO lock as writer; otherwise they can only
1024                         * transition from FALSE to TRUE.  This ensures that
1025                         * any zio looking at these values can assume that
1026                         * failures persist for the life of the I/O.  That's
1027                         * important because when a device has intermittent
1028                         * connectivity problems, we want to ensure that
1029                         * they're ascribed to the device (ENXIO) and not
1030                         * the zio (EIO).
1031                         *
1032                         * Since we hold SCL_ZIO as writer here, clear both
1033                         * values so the probe can reevaluate from first
1034                         * principles.
1035                         */
1036                        vps->vps_flags |= ZIO_FLAG_CONFIG_WRITER;
1037                        vd->vdev_cant_read = B_FALSE;
1038                        vd->vdev_cant_write = B_FALSE;
1039                }

1041                vd->vdev_probe_zio = pio = zio_null(NULL, spa, vd,
1042                    vdev_probe_done, vps,
1043                    vps->vps_flags | ZIO_FLAG_DONT_PROPAGATE);

1045                /*
1046                 * We can't change the vdev state in this context, so we
1047                 * kick off an async task to do it on our behalf.
1048                 */
1049                if (zio != NULL) {
1050                        vd->vdev_probe_wanted = B_TRUE;
1051                        spa_async_request(spa, SPA_ASYNC_PROBE);
1052                }
1053        }

1055        if (zio != NULL)
1056                zio_add_child(zio, pio);

1058        mutex_exit(&vd->vdev_probe_lock);

1060        if (vps == NULL) {
1061                ASSERT(zio != NULL);
1062                return (NULL);
1063        }

1065        for (int l = 1; l < VDEV_LABELS; l++) {
1066                zio_nowait(zio_read_phys(pio, vd,
1067                    vdev_label_offset(vd->vdev_psize, l,
1068                    offsetof(vdev_label_t, vl_pad2)),
1069                    VDEV_PAD_SIZE, zio_buf_alloc(VDEV_PAD_SIZE),
1070                    ZIO_CHECKSUM_OFF, vdev_probe_done, vps,
1071                    ZIO_PRIORITY_SYNC_READ, vps->vps_flags, B_TRUE));
1072        }

1074        if (zio == NULL)
1075                return (pio);

1077        zio_nowait(pio);
1078        return (NULL);
1079 }

1081 static void
1082 vdev_open_child(void *arg)
1083 {
1084         vdev_t *vd = arg;

1086         vd->vdev_open_thread = curthread;
1087         vd->vdev_open_error = vdev_open(vd);
```

```
1088         vd->vdev_open_thread = NULL;
1089 }

1091 boolean_t
1092 vdev_uses_zvols(vdev_t *vd)
1093 {
1094         if (vd->vdev_path && strncmp(vd->vdev_path, ZVOL_DIR,
1095             strlen(ZVOL_DIR)) == 0)
1096                 return (B_TRUE);
1097         for (int c = 0; c < vd->vdev_children; c++)
1098                 if (vdev_uses_zvols(vd->vdev_child[c]))
1099                         return (B_TRUE);
1100         return (B_FALSE);
1101 }

1103 void
1104 vdev_open_children(vdev_t *vd)
1105 {
1106         taskq_t *tq;
1107         int children = vd->vdev_children;

1109         /*
1110          * in order to handle pools on top of zvols, do the opens
1111          * in a single thread so that the same thread holds the
1112          * spa_namespace_lock
1113          */
1114         if (vdev_uses_zvols(vd)) {
1115                 for (int c = 0; c < children; c++)
1116                         vd->vdev_child[c]->vdev_open_error =
1117                             vdev_open(vd->vdev_child[c]);
1118                 return;
1119         }
1120         tq = taskq_create("vdev_open", children, minclsyspri,
1121             children, children, TASKQ_PREPOPULATE);

1123         for (int c = 0; c < children; c++)
1124                 VERIFY(taskq_dispatch(tq, vdev_open_child, vd->vdev_child[c],
1125                     TQ_SLEEP) != NULL);

1127         taskq_destroy(tq);
1128 }

1130 /*
1131  * Prepare a virtual device for access.
1132  */
1133 int
1134 vdev_open(vdev_t *vd)
1135 {
1136         spa_t *spa = vd->vdev_spa;
1137         int error;
1138         uint64_t osize = 0;
1139         uint64_t max_osize = 0;
1140         uint64_t asize, max_asize, psize;
1141         uint64_t ashift = 0;

1143         ASSERT(vd->vdev_open_thread == curthread ||
1144             spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
1145         ASSERT(vd->vdev_state == VDEV_STATE_CLOSED ||
1146             vd->vdev_state == VDEV_STATE_CANT_OPEN ||
1147             vd->vdev_state == VDEV_STATE_OFFLINE);

1149         vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
1150         vd->vdev_cant_read = B_FALSE;
1151         vd->vdev_cant_write = B_FALSE;
1152         vd->vdev_min_asize = vdev_get_min_asize(vd);
```



```
1154         /*
1155          * If this vdev is not removed, check its fault status.  If it's
1156          * faulted, bail out of the open.
1157          */
1158         if (!vd->vdev_removed && vd->vdev_faulted) {
1159                 ASSERT(vd->vdev_children == 0);
1160                 ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
1161                     vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
1162                 vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1163                     vd->vdev_label_aux);
1164                 return (SET_ERROR(ENXIO));
1165         } else if (vd->vdev_offline) {
1166                 ASSERT(vd->vdev_children == 0);
1167                 vdev_set_state(vd, B_TRUE, VDEV_STATE_OFFLINE, VDEV_AUX_NONE);
1168                 return (SET_ERROR(ENXIO));
1169         }

1171         error = vd->vdev_ops->vdev_op_open(vd, &osize, &max_osize, &ashift);

1173         /*
1174          * Reset the vdev_reopening flag so that we actually close
1175          * the vdev on error.
1176          */
1177         vd->vdev_reopening = B_FALSE;
1178         if (zio_injection_enabled && error == 0)
1179                 error = zio_handle_device_injection(vd, NULL, ENXIO);

1181         if (error) {
1182                 if (vd->vdev_removed &&
1183                     vd->vdev_stat.vs_aux != VDEV_AUX_OPEN_FAILED)
1184                         vd->vdev_removed = B_FALSE;

1186                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1187                     vd->vdev_stat.vs_aux);
1188                 return (error);
1189         }

1191         vd->vdev_removed = B_FALSE;

1193         /*
1194          * Recheck the faulted flag now that we have confirmed that
1195          * the vdev is accessible.  If we're faulted, bail.
1196          */
1197         if (vd->vdev_faulted) {
1198                 ASSERT(vd->vdev_children == 0);
1199                 ASSERT(vd->vdev_label_aux == VDEV_AUX_ERR_EXCEEDED ||
1200                     vd->vdev_label_aux == VDEV_AUX_EXTERNAL);
1201                 vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1202                     vd->vdev_label_aux);
1203                 return (SET_ERROR(ENXIO));
1204         }

1206         if (vd->vdev_degraded) {
1207                 ASSERT(vd->vdev_children == 0);
1208                 vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
1209                     VDEV_AUX_ERR_EXCEEDED);
1210         } else {
1211                 vdev_set_state(vd, B_TRUE, VDEV_STATE_HEALTHY, 0);
1212         }

1214         /*
1215          * For hole or missing vdevs we just return success.
1216          */
1217         if (vd->vdev_ishole || vd->vdev_ops == &vdev_missing_ops)
1218                 return (0);
```

```
1220         for (int c = 0; c < vd->vdev_children; c++) {
1221                 if (vd->vdev_child[c]->vdev_state != VDEV_STATE_HEALTHY) {
1222                         vdev_set_state(vd, B_TRUE, VDEV_STATE_DEGRADED,
1223                             VDEV_AUX_NONE);
1224                         break;
1225                 }
1226         }

1228         osize = P2ALIGN(osize, (uint64_t)sizeof (vdev_label_t));
1229         max_osize = P2ALIGN(max_osize, (uint64_t)sizeof (vdev_label_t));

1231         if (vd->vdev_children == 0) {
1232                 if (osize < SPA_MINDEVSIZE) {
1233                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1234                             VDEV_AUX_TOO_SMALL);
1235                         return (SET_ERROR(EOVERFLOW));
1236                 }
1237                 psize = osize;
1238                 asize = osize - (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE);
1239                 max_asize = max_osize - (VDEV_LABEL_START_SIZE +
1240                     VDEV_LABEL_END_SIZE);
1241         } else {
1242                 if (vd->vdev_parent != NULL && osize < SPA_MINDEVSIZE -
1243                     (VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE)) {
1244                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1245                             VDEV_AUX_TOO_SMALL);
1246                         return (SET_ERROR(EOVERFLOW));
1247                 }
1248                 psize = 0;
1249                 asize = osize;
1250                 max_asize = max_osize;
1251         }

1253         vd->vdev_psize = psize;

1255         /*
1256          * Make sure the allocatable size hasn't shrunk.
1257          */
1258         if (asize < vd->vdev_min_asize) {
1259                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1260                     VDEV_AUX_BAD_LABEL);
1261                 return (SET_ERROR(EINVAL));
1262         }

1264         if (vd->vdev_asize == 0) {
1265                 /*
1266                  * This is the first-ever open, so use the computed values.
1267                  * For testing purposes, a higher ashift can be requested.
1268                  */
1269                 vd->vdev_asize = asize;
1270                 vd->vdev_max_asize = max_asize;
1271                 vd->vdev_ashift = MAX(ashift, vd->vdev_ashift);
1272         } else {
1273                 /*
1274                  * Detect if the alignment requirement has increased.
1275                  * We don't want to make the pool unavailable, just
1276                  * issue a warning instead.
1277                  */
1278                 if (ashift > vd->vdev_top->vdev_ashift &&
1279                     vd->vdev_ops->vdev_op_leaf) {
1280                         cmn_err(CE_WARN,
1281                             "Disk, '%s', has a block alignment that is "
1282                             "larger than the pool's alignment\n",
1283                             vd->vdev_path);
1284                 }
1285                 vd->vdev_max_asize = max_asize;
```



```
1286         }

1288         /*
1289          * If all children are healthy and the asize has increased,
1290          * then we've experienced dynamic LUN growth.  If automatic
1291          * expansion is enabled then use the additional space.
1292          */
1293         if (vd->vdev_state == VDEV_STATE_HEALTHY && asize > vd->vdev_asize &&
1294             (vd->vdev_expanding || spa->spa_autoexpand))
1295                 vd->vdev_asize = asize;

1297         vdev_set_min_asize(vd);

1299         /*
1300          * Ensure we can issue some IO before declaring the
1301          * vdev open for business.
1302          */
1303         if (vd->vdev_ops->vdev_op_leaf &&
1304             (error = zio_wait(vdev_probe(vd, NULL))) != 0) {
1305                 vdev_set_state(vd, B_TRUE, VDEV_STATE_FAULTED,
1306                     VDEV_AUX_ERR_EXCEEDED);
1307                 return (error);
1308         }

1310         /*
1311          * If a leaf vdev has a DTL, and seems healthy, then kick off a
1312          * resilver.  But don't do this if we are doing a reopen for a scrub,
1313          * since this would just restart the scrub we are already doing.
1314          */
1315         if (vd->vdev_ops->vdev_op_leaf && !spa->spa_scrub_reopen &&
1316             vdev_resilver_needed(vd, NULL, NULL))
1317                 spa_async_request(spa, SPA_ASYNC_RESILVER);

1319         return (0);
1320 }

1322 /*
1323  * Called once the vdevs are all opened, this routine validates the label
1324  * contents.  This needs to be done before vdev_load() so that we don't
1325  * inadvertently do repair I/Os to the wrong device.
1326  *
1327  * If 'strict' is false ignore the spa guid check. This is necessary because
1328  * if the machine crashed during a re-guid the new guid might have been written
1329  * to all of the vdev labels, but not the cached config. The strict check
1330  * will be performed when the pool is opened again using the mos config.
1331  *
1332  * This function will only return failure if one of the vdevs indicates that it
1333  * has since been destroyed or exported.  This is only possible if
1334  * /etc/zfs/zpool.cache was readonly at the time.  Otherwise, the vdev state
1335  * will be updated but the function will return 0.
1336  */
1337 int
1338 vdev_validate(vdev_t *vd, boolean_t strict)
1339 {
1340         spa_t *spa = vd->vdev_spa;
1341         nvlist_t *label;
1342         uint64_t guid = 0, top_guid;
1343         uint64_t state;

1345         for (int c = 0; c < vd->vdev_children; c++)
1346                 if (vdev_validate(vd->vdev_child[c], strict) != 0)
1347                         return (SET_ERROR(EBADF));

1349         /*
1350          * If the device has already failed, or was marked offline, don't do
1351          * any further validation.  Otherwise, label I/O will fail and we will
```

```
1352          * overwrite the previous state.
1353          */
1354         if (vd->vdev_ops->vdev_op_leaf && vdev_readable(vd)) {
1355                 uint64_t aux_guid = 0;
1356                 nvlist_t *nvl;
1357                 uint64_t txg = spa_last_synced_txg(spa) != 0 ?
1358                     spa_last_synced_txg(spa) : -1ULL;

1360                 if ((label = vdev_label_read_config(vd, txg)) == NULL) {
1361                         vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
1362                             VDEV_AUX_BAD_LABEL);
1363                         return (0);
1364                 }

1366                 /*
1367                  * Determine if this vdev has been split off into another
1368                  * pool.  If so, then refuse to open it.
1369                  */
1370                 if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_SPLIT_GUID,
1371                     &aux_guid) == 0 && aux_guid == spa_guid(spa)) {
1372                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1373                             VDEV_AUX_SPLIT_POOL);
1374                         nvlist_free(label);
1375                         return (0);
1376                 }

1378                 if (strict && (nvlist_lookup_uint64(label,
1379                     ZPOOL_CONFIG_POOL_GUID, &guid) != 0 ||
1380                     guid != spa_guid(spa))) {
1381                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1382                             VDEV_AUX_CORRUPT_DATA);
1383                         nvlist_free(label);
1384                         return (0);
1385                 }

1387                 if (nvlist_lookup_nvlist(label, ZPOOL_CONFIG_VDEV_TREE, &nvl)
1388                     != 0 || nvlist_lookup_uint64(nvl, ZPOOL_CONFIG_ORIG_GUID,
1389                     &aux_guid) != 0)
1390                         aux_guid = 0;

1392                 /*
1393                  * If this vdev just became a top-level vdev because its
1394                  * sibling was detached, it will have adopted the parent's
1395                  * vdev guid -- but the label may or may not be on disk yet.
1396                  * Fortunately, either version of the label will have the
1397                  * same top guid, so if we're a top-level vdev, we can
1398                  * safely compare to that instead.
1399                  *
1400                  * If we split this vdev off instead, then we also check the
1401                  * original pool's guid.  We don't want to consider the vdev
1402                  * corrupt if it is partway through a split operation.
1403                  */
1404                 if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID,
1405                     &guid) != 0 ||
1406                     nvlist_lookup_uint64(label, ZPOOL_CONFIG_TOP_GUID,
1407                     &top_guid) != 0 ||
1408                     ((vd->vdev_guid != guid && vd->vdev_guid != aux_guid) &&
1409                     (vd->vdev_guid != top_guid || vd != vd->vdev_top))) {
1410                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1411                             VDEV_AUX_CORRUPT_DATA);
1412                         nvlist_free(label);
1413                         return (0);
1414                 }

1416                 if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE,
1417                     &state) != 0) {
```



```
1418                         vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
1419                             VDEV_AUX_CORRUPT_DATA);
1420                         nvlist_free(label);
1421                         return (0);
1422                 }

1424                 nvlist_free(label);

1426                 /*
1427                  * If this is a verbatim import, no need to check the
1428                  * state of the pool.
1429                  */
1430                 if (!(spa->spa_import_flags & ZFS_IMPORT_VERBATIM) &&
1431                     spa_load_state(spa) == SPA_LOAD_OPEN &&
1432                     state != POOL_STATE_ACTIVE)
1433                         return (SET_ERROR(EBADF));

1435                 /*
1436                  * If we were able to open and validate a vdev that was
1437                  * previously marked permanently unavailable, clear that state
1438                  * now.
1439                  */
1440                 if (vd->vdev_not_present)
1441                         vd->vdev_not_present = 0;
1442         }

1444         return (0);
1445 }

1447 /*
1448  * Close a virtual device.
1449  */
1450 void
1451 vdev_close(vdev_t *vd)
1452 {
1453         spa_t *spa = vd->vdev_spa;
1454         vdev_t *pvd = vd->vdev_parent;

1456         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1458         /*
1459          * If our parent is reopening, then we are as well, unless we are
1460          * going offline.
1461          */
1462         if (pvd != NULL && pvd->vdev_reopening)
1463                 vd->vdev_reopening = (pvd->vdev_reopening && !vd->vdev_offline);

1465         vd->vdev_ops->vdev_op_close(vd);

1467         vdev_cache_purge(vd);

1469         /*
1470          * We record the previous state before we close it, so that if we are
1471          * doing a reopen(), we don't generate FMA ereports if we notice that
1472          * it's still faulted.
1473          */
1474         vd->vdev_prevstate = vd->vdev_state;

1476         if (vd->vdev_offline)
1477                 vd->vdev_state = VDEV_STATE_OFFLINE;
1478         else
1479                 vd->vdev_state = VDEV_STATE_CLOSED;
1480         vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
1481 }

1483 void
```

```
1484 vdev_hold(vdev_t *vd)
1485 {
1486         spa_t *spa = vd->vdev_spa;

1488         ASSERT(spa_is_root(spa));
1489         if (spa->spa_state == POOL_STATE_UNINITIALIZED)
1490                 return;

1492         for (int c = 0; c < vd->vdev_children; c++)
1493                 vdev_hold(vd->vdev_child[c]);

1495         if (vd->vdev_ops->vdev_op_leaf)
1496                 vd->vdev_ops->vdev_op_hold(vd);
1497 }

1499 void
1500 vdev_rele(vdev_t *vd)
1501 {
1502         spa_t *spa = vd->vdev_spa;

1504         ASSERT(spa_is_root(spa));
1505         for (int c = 0; c < vd->vdev_children; c++)
1506                 vdev_rele(vd->vdev_child[c]);

1508         if (vd->vdev_ops->vdev_op_leaf)
1509                 vd->vdev_ops->vdev_op_rele(vd);
1510 }

1512 /*
1513  * Reopen all interior vdevs and any unopened leaves.  We don't actually
1514  * reopen leaf vdevs which had previously been opened as they might deadlock
1515  * on the spa_config_lock.  Instead we only obtain the leaf's physical size.
1516  * If the leaf has never been opened then open it, as usual.
1517  */
1518 void
1519 vdev_reopen(vdev_t *vd)
1520 {
1521         spa_t *spa = vd->vdev_spa;

1523         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

1525         /* set the reopening flag unless we're taking the vdev offline */
1526         vd->vdev_reopening = !vd->vdev_offline;
1527         vdev_close(vd);
1528         (void) vdev_open(vd);

1530         /*
1531          * Call vdev_validate() here to make sure we have the same device.
1532          * Otherwise, a device with an invalid label could be successfully
1533          * opened in response to vdev_reopen().
1534          */
1535         if (vd->vdev_aux) {
1536                 (void) vdev_validate_aux(vd);
1537                 if (vdev_readable(vd) && vdev_writeable(vd) &&
1538                     vd->vdev_aux == &spa->spa_l2cache &&
1539                     !l2arc_vdev_present(vd))
1540                         l2arc_add_vdev(spa, vd);
1541         } else {
1542                 (void) vdev_validate(vd, B_TRUE);
1543         }

1545         /*
1546          * Reassess parent vdev's health.
1547          */
1548         vdev_propagate_state(vd);
1549 }
```



```
1551 int
1552 vdev_create(vdev_t *vd, uint64_t txg, boolean_t isreplacing)
1553 {
1554         int error;

1556         /*
1557          * Normally, partial opens (e.g. of a mirror) are allowed.
1558          * For a create, however, we want to fail the request if
1559          * there are any components we can't open.
1560          */
1561         error = vdev_open(vd);

1563         if (error || vd->vdev_state != VDEV_STATE_HEALTHY) {
1564                 vdev_close(vd);
1565                 return (error ? error : ENXIO);
1566         }

1568         /*
1569          * Recursively load DTLs and initialize all labels.
1570          */
1571         if ((error = vdev_dtl_load(vd)) != 0 ||
1572             (error = vdev_label_init(vd, txg, isreplacing ?
1573             VDEV_LABEL_REPLACE : VDEV_LABEL_CREATE)) != 0) {
1574                 vdev_close(vd);
1575                 return (error);
1576         }

1578         return (0);
1579 }

1581 void
1582 vdev_metaslab_set_size(vdev_t *vd)
1583 {
1584         /*
1585          * Aim for roughly metaslabs_per_vdev (default 200) metaslabs per vdev.
1586          */
1587         vd->vdev_ms_shift = highbit64(vd->vdev_asize / metaslabs_per_vdev);
1588         vd->vdev_ms_shift = MAX(vd->vdev_ms_shift, SPA_MAXBLOCKSHIFT);
1589 }

1591 void
1592 vdev_dirty(vdev_t *vd, int flags, void *arg, uint64_t txg)
1593 {
1594         ASSERT(vd == vd->vdev_top);
1595         ASSERT(!vd->vdev_ishole);
1596         ASSERT(ISP2(flags));
1597         ASSERT(spa_writeable(vd->vdev_spa));

1599         if (flags & VDD_METASLAB)
1600                 (void) txg_list_add(&vd->vdev_ms_list, arg, txg);

1602         if (flags & VDD_DTL)
1603                 (void) txg_list_add(&vd->vdev_dtl_list, arg, txg);

1605         (void) txg_list_add(&vd->vdev_spa->spa_vdev_txg_list, vd, txg);
1606 }

1608 void
1609 vdev_dirty_leaves(vdev_t *vd, int flags, uint64_t txg)
1610 {
1611         for (int c = 0; c < vd->vdev_children; c++)
1612                 vdev_dirty_leaves(vd->vdev_child[c], flags, txg);

1614         if (vd->vdev_ops->vdev_op_leaf)
1615                 vdev_dirty(vd->vdev_top, flags, vd, txg);
```

```
1616 }

1618 /*
1619  * DTLs.
1620  *
1621  * A vdev's DTL (dirty time log) is the set of transaction groups for which
1622  * the vdev has less than perfect replication.  There are four kinds of DTL:
1623  *
1624  * DTL_MISSING: txgs for which the vdev has no valid copies of the data
1625  *
1626  * DTL_PARTIAL: txgs for which data is available, but not fully replicated
1627  *
1628  * DTL_SCRUB: the txgs that could not be repaired by the last scrub; upon
1629  *      scrub completion, DTL_SCRUB replaces DTL_MISSING in the range of
1630  *      txgs that was scrubbed.
1631  *
1632  * DTL_OUTAGE: txgs which cannot currently be read, whether due to
1633  *      persistent errors or just some device being offline.
1634  *      Unlike the other three, the DTL_OUTAGE map is not generally
1635  *      maintained; it's only computed when needed, typically to
1636  *      determine whether a device can be detached.
1637  *
1638  * For leaf vdevs, DTL_MISSING and DTL_PARTIAL are identical: the device
1639  * either has the data or it doesn't.
1640  *
1641  * For interior vdevs such as mirror and RAID-Z the picture is more complex.
1642  * A vdev's DTL_PARTIAL is the union of its children's DTL_PARTIALs, because
1643  * if any child is less than fully replicated, then so is its parent.
1644  * A vdev's DTL_MISSING is a modified union of its children's DTL_MISSINGs,
1645  * comprising only those txgs which appear in 'maxfaults' or more children;
1646  * those are the txgs we don't have enough replication to read.  For example,
1647  * double-parity RAID-Z can tolerate up to two missing devices (maxfaults == 2);
1648  * thus, its DTL_MISSING consists of the set of txgs that appear in more than
1649  * two child DTL_MISSING maps.
1650  *
1651  * It should be clear from the above that to compute the DTLs and outage maps
1652  * for all vdevs, it suffices to know just the leaf vdevs' DTL_MISSING maps.
1653  * Therefore, that is all we keep on disk.  When loading the pool, or after
1654  * a configuration change, we generate all other DTLs from first principles.
1655  */
1656 void
1657 vdev_dtl_dirty(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
1658 {
1659         range_tree_t *rt = vd->vdev_dtl[t];

1661         ASSERT(t < DTL_TYPES);
1662         ASSERT(vd != vd->vdev_spa->spa_root_vdev);
1663         ASSERT(spa_writeable(vd->vdev_spa));

1665         mutex_enter(rt->rt_lock);
1666         if (!range_tree_contains(rt, txg, size))
1667                 range_tree_add(rt, txg, size);
1668         mutex_exit(rt->rt_lock);
1669 }

1671 boolean_t
1672 vdev_dtl_contains(vdev_t *vd, vdev_dtl_type_t t, uint64_t txg, uint64_t size)
1673 {
1674         range_tree_t *rt = vd->vdev_dtl[t];
1675         boolean_t dirty = B_FALSE;

1677         ASSERT(t < DTL_TYPES);
1678         ASSERT(vd != vd->vdev_spa->spa_root_vdev);

1680         mutex_enter(rt->rt_lock);
1681         if (range_tree_space(rt) != 0)
```



```
1682                 dirty = range_tree_contains(rt, txg, size);
1683         mutex_exit(rt->rt_lock);

1685         return (dirty);
1686 }

1688 boolean_t
1689 vdev_dtl_empty(vdev_t *vd, vdev_dtl_type_t t)
1690 {
1691         range_tree_t *rt = vd->vdev_dtl[t];
1692         boolean_t empty;

1694         mutex_enter(rt->rt_lock);
1695         empty = (range_tree_space(rt) == 0);
1696         mutex_exit(rt->rt_lock);

1698         return (empty);
1699 }

1701 /*
1702  * Returns the lowest txg in the DTL range.
1703  */
1704 static uint64_t
1705 vdev_dtl_min(vdev_t *vd)
1706 {
1707         range_seg_t *rs;

1709         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1710         ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
1711         ASSERT0(vd->vdev_children);

1713         rs = avl_first(&vd->vdev_dtl[DTL_MISSING]->rt_root);
1714         return (rs->rs_start - 1);
1715 }

1717 /*
1718  * Returns the highest txg in the DTL.
1719  */
1720 static uint64_t
1721 vdev_dtl_max(vdev_t *vd)
1722 {
1723         range_seg_t *rs;

1725         ASSERT(MUTEX_HELD(&vd->vdev_dtl_lock));
1726         ASSERT3U(range_tree_space(vd->vdev_dtl[DTL_MISSING]), !=, 0);
1727         ASSERT0(vd->vdev_children);

1729         rs = avl_last(&vd->vdev_dtl[DTL_MISSING]->rt_root);
1730         return (rs->rs_end);
1731 }

1733 /*
1734  * Determine if a resilvering vdev should remove any DTL entries from
1735  * its range. If the vdev was resilvering for the entire duration of the
1736  * scan then it should excise that range from its DTLs. Otherwise, this
1737  * vdev is considered partially resilvered and should leave its DTL
1738  * entries intact. The comment in vdev_dtl_reassess() describes how we
1739  * excise the DTLs.
1740  */
1741 static boolean_t
1742 vdev_dtl_should_excise(vdev_t *vd)
1743 {
1744         spa_t *spa = vd->vdev_spa;
1745         dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;

1747         ASSERT0(scn->scn_phys.scn_errors);
```

```
1748         ASSERT0(vd->vdev_children);
1749
1750         if (vd->vdev_resilver_txg == 0 ||
1751             range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0)
1752                 return (B_TRUE);
1753
1754         /*
1755          * When a resilver is initiated the scan will assign the scn_max_txg
1756          * value to the highest txg value that exists in all DTLs. If this
1757          * device's max DTL is not part of this scan (i.e. it is not in
1758          * the range (scn_min_txg, scn_max_txg] then it is not eligible
1759          * for excision.
1760          */
1761         if (vdev_dtl_max(vd) <= scn->scn_phys.scn_max_txg) {
1762                 ASSERT3U(scn->scn_phys.scn_min_txg, <=, vdev_dtl_min(vd));
1763                 ASSERT3U(scn->scn_phys.scn_min_txg, <, vd->vdev_resilver_txg);
1764                 ASSERT3U(vd->vdev_resilver_txg, <=, scn->scn_phys.scn_max_txg);
1765                 return (B_TRUE);
1766         }
1767         return (B_FALSE);
1768 }
1769
1770 /*
1771  * Reassess DTLs after a config change or scrub completion.
1772  */
1773 void
1774 vdev_dtl_reassess(vdev_t *vd, uint64_t txg, uint64_t scrub_txg, int scrub_done)
1775 {
1776         spa_t *spa = vd->vdev_spa;
1777         avl_tree_t reftree;
1778         int minref;
1779
1780         ASSERT(spa_config_held(spa, SCL_ALL, RW_READER) != 0);
1781
1782         for (int c = 0; c < vd->vdev_children; c++)
1783                 vdev_dtl_reassess(vd->vdev_child[c], txg,
1784                     scrub_txg, scrub_done);
1785
1786         if (vd == spa->spa_root_vdev || vd->vdev_ishole || vd->vdev_aux)
1787                 return;
1788
1789         if (vd->vdev_ops->vdev_op_leaf) {
1790                 dsl_scan_t *scn = spa->spa_dsl_pool->dp_scan;
1791
1792                 mutex_enter(&vd->vdev_dtl_lock);
1793
1794                 /*
1795                  * If we've completed a scan cleanly then determine
1796                  * if this vdev should remove any DTLs. We only want to
1797                  * excise regions on vdevs that were available during
1798                  * the entire duration of this scan.
1799                  */
1800                 if (scrub_txg != 0 &&
1801                     (spa->spa_scrub_started ||
1802                     (scn != NULL && scn->scn_phys.scn_errors == 0)) &&
1803                     vdev_dtl_should_excise(vd)) {
1804                         /*
1805                          * We completed a scrub up to scrub_txg.  If we
1806                          * did it without rebooting, then the scrub dtl
1807                          * will be valid, so excise the old region and
1808                          * fold in the scrub dtl.  Otherwise, leave the
1809                          * dtl as-is if there was an error.
1810                          *
1811                          * There's little trick here: to excise the beginning
1812                          * of the DTL_MISSING map, we put it into a reference
1813                          * tree and then add a segment with refcnt -1 that
```



```
1814                          * covers the range [0, scrub_txg).  This means
1815                          * that each txg in that range has refcnt -1 or 0.
1816                          * We then add DTL_SCRUB with a refcnt of 2, so that
1817                          * entries in the range [0, scrub_txg) will have a
1818                          * positive refcnt -- either 1 or 2.  We then convert
1819                          * the reference tree into the new DTL_MISSING map.
1820                          */
1821                         space_reftree_create(&reftree);
1822                         space_reftree_add_map(&reftree,
1823                             vd->vdev_dtl[DTL_MISSING], 1);
1824                         space_reftree_add_seg(&reftree, 0, scrub_txg, -1);
1825                         space_reftree_add_map(&reftree,
1826                             vd->vdev_dtl[DTL_SCRUB], 2);
1827                         space_reftree_generate_map(&reftree,
1828                             vd->vdev_dtl[DTL_MISSING], 1);
1829                         space_reftree_destroy(&reftree);
1830                 }
1831                 range_tree_vacate(vd->vdev_dtl[DTL_PARTIAL], NULL, NULL);
1832                 range_tree_walk(vd->vdev_dtl[DTL_MISSING],
1833                     range_tree_add, vd->vdev_dtl[DTL_PARTIAL]);
1834                 if (scrub_done)
1835                         range_tree_vacate(vd->vdev_dtl[DTL_SCRUB], NULL, NULL);
1836                 range_tree_vacate(vd->vdev_dtl[DTL_OUTAGE], NULL, NULL);
1837                 if (!vdev_readable(vd))
1838                         range_tree_add(vd->vdev_dtl[DTL_OUTAGE], 0, -1ULL);
1839                 else
1840                         range_tree_walk(vd->vdev_dtl[DTL_MISSING],
1841                             range_tree_add, vd->vdev_dtl[DTL_OUTAGE]);
1842
1843                 /*
1844                  * If the vdev was resilvering and no longer has any
1845                  * DTLs then reset its resilvering flag.
1846                  */
1847                 if (vd->vdev_resilver_txg != 0 &&
1848                     range_tree_space(vd->vdev_dtl[DTL_MISSING]) == 0 &&
1849                     range_tree_space(vd->vdev_dtl[DTL_OUTAGE]) == 0)
1850                         vd->vdev_resilver_txg = 0;
1851
1852                 mutex_exit(&vd->vdev_dtl_lock);
1853
1854                 if (txg != 0)
1855                         vdev_dirty(vd->vdev_top, VDD_DTL, vd, txg);
1856                 return;
1857         }
1858
1859         mutex_enter(&vd->vdev_dtl_lock);
1860         for (int t = 0; t < DTL_TYPES; t++) {
1861                 /* account for child's outage in parent's missing map */
1862                 int s = (t == DTL_MISSING) ? DTL_OUTAGE: t;
1863                 if (t == DTL_SCRUB)
1864                         continue;                       /* leaf vdevs only */
1865                 if (t == DTL_PARTIAL)
1866                         minref = 1;                     /* i.e. non-zero */
1867                 else if (vd->vdev_nparity != 0)
1868                         minref = vd->vdev_nparity + 1;  /* RAID-Z */
1869                 else
1870                         minref = vd->vdev_children;     /* any kind of mirror */
1871                 space_reftree_create(&reftree);
1872                 for (int c = 0; c < vd->vdev_children; c++) {
1873                         vdev_t *cvd = vd->vdev_child[c];
1874                         mutex_enter(&cvd->vdev_dtl_lock);
1875                         space_reftree_add_map(&reftree, cvd->vdev_dtl[s], 1);
1876                         mutex_exit(&cvd->vdev_dtl_lock);
1877                 }
1878                 space_reftree_generate_map(&reftree, vd->vdev_dtl[t], minref);
1879                 space_reftree_destroy(&reftree);
```

```
1880         }
1881         mutex_exit(&vd->vdev_dtl_lock);
1882 }

1884 int
1885 vdev_dtl_load(vdev_t *vd)
1886 {
1887         spa_t *spa = vd->vdev_spa;
1888         objset_t *mos = spa->spa_meta_objset;
1889         int error = 0;

1891         if (vd->vdev_ops->vdev_op_leaf && vd->vdev_dtl_object != 0) {
1892                 ASSERT(!vd->vdev_ishole);

1894                 error = space_map_open(&vd->vdev_dtl_sm, mos,
1895                     vd->vdev_dtl_object, 0, -1ULL, 0, &vd->vdev_dtl_lock);
1896                 if (error)
1897                         return (error);
1898                 ASSERT(vd->vdev_dtl_sm != NULL);

1900                 mutex_enter(&vd->vdev_dtl_lock);

1902                 /*
1903                  * Now that we've opened the space_map we need to update
1904                  * the in-core DTL.
1905                  */
1906                 space_map_update(vd->vdev_dtl_sm);

1908                 error = space_map_load(vd->vdev_dtl_sm,
1909                     vd->vdev_dtl[DTL_MISSING], SM_ALLOC);
1910                 mutex_exit(&vd->vdev_dtl_lock);

1912                 return (error);
1913         }

1915         for (int c = 0; c < vd->vdev_children; c++) {
1916                 error = vdev_dtl_load(vd->vdev_child[c]);
1917                 if (error != 0)
1918                         break;
1919         }

1921         return (error);
1922 }

1924 void
1925 vdev_dtl_sync(vdev_t *vd, uint64_t txg)
1926 {
1927         spa_t *spa = vd->vdev_spa;
1928         range_tree_t *rt = vd->vdev_dtl[DTL_MISSING];
1929         objset_t *mos = spa->spa_meta_objset;
1930         range_tree_t *rtsync;
1931         kmutex_t rtlock;
1932         dmu_tx_t *tx;
1933         uint64_t object = space_map_object(vd->vdev_dtl_sm);

1935         ASSERT(!vd->vdev_ishole);
1936         ASSERT(vd->vdev_ops->vdev_op_leaf);

1938         tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);

1940         if (vd->vdev_detached || vd->vdev_top->vdev_removing) {
1941                 mutex_enter(&vd->vdev_dtl_lock);
1942                 space_map_free(vd->vdev_dtl_sm, tx);
1943                 space_map_close(vd->vdev_dtl_sm);
1944                 vd->vdev_dtl_sm = NULL;
1945                 mutex_exit(&vd->vdev_dtl_lock);
```



```
1946                 dmu_tx_commit(tx);
1947                 return;
1948         }

1950         if (vd->vdev_dtl_sm == NULL) {
1951                 uint64_t new_object;

1953                 new_object = space_map_alloc(mos, tx);
1954                 VERIFY3U(new_object, !=, 0);

1956                 VERIFY0(space_map_open(&vd->vdev_dtl_sm, mos, new_object,
1957                     0, -1ULL, 0, &vd->vdev_dtl_lock));
1958                 ASSERT(vd->vdev_dtl_sm != NULL);
1959         }

1961         mutex_init(&rtlock, NULL, MUTEX_DEFAULT, NULL);

1963         rtsync = range_tree_create(NULL, NULL, &rtlock);

1965         mutex_enter(&rtlock);

1967         mutex_enter(&vd->vdev_dtl_lock);
1968         range_tree_walk(rt, range_tree_add, rtsync);
1969         mutex_exit(&vd->vdev_dtl_lock);

1971         space_map_truncate(vd->vdev_dtl_sm, tx);
1972         space_map_write(vd->vdev_dtl_sm, rtsync, SM_ALLOC, tx);
1973         range_tree_vacate(rtsync, NULL, NULL);

1975         range_tree_destroy(rtsync);

1977         mutex_exit(&rtlock);
1978         mutex_destroy(&rtlock);

1980         /*
1981          * If the object for the space map has changed then dirty
1982          * the top level so that we update the config.
1983          */
1984         if (object != space_map_object(vd->vdev_dtl_sm)) {
1985                 zfs_dbgmsg("txg %llu, spa %s, DTL old object %llu, "
1986                     "new object %llu", txg, spa_name(spa), object,
1987                     space_map_object(vd->vdev_dtl_sm));
1988                 vdev_config_dirty(vd->vdev_top);
1989         }

1991         dmu_tx_commit(tx);

1993         mutex_enter(&vd->vdev_dtl_lock);
1994         space_map_update(vd->vdev_dtl_sm);
1995         mutex_exit(&vd->vdev_dtl_lock);
1996 }

1998 /*
1999  * Determine whether the specified vdev can be offlined/detached/removed
2000  * without losing data.
2001  */
2002 boolean_t
2003 vdev_dtl_required(vdev_t *vd)
2004 {
2005         spa_t *spa = vd->vdev_spa;
2006         vdev_t *tvd = vd->vdev_top;
2007         uint8_t cant_read = vd->vdev_cant_read;
2008         boolean_t required;

2010         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
```

```
2012         if (vd == spa->spa_root_vdev || vd == tvd)
2013                 return (B_TRUE);
2015         /*
2016          * Temporarily mark the device as unreadable, and then determine
2017          * whether this results in any DTL outages in the top-level vdev.
2018          * If not, we can safely offline/detach/remove the device.
2019          */
2020         vd->vdev_cant_read = B_TRUE;
2021         vdev_dtl_reassess(tvd, 0, 0, B_FALSE);
2022         required = !vdev_dtl_empty(tvd, DTL_OUTAGE);
2023         vd->vdev_cant_read = cant_read;
2024         vdev_dtl_reassess(tvd, 0, 0, B_FALSE);
2026         if (!required && zio_injection_enabled)
2027                 required = !!zio_handle_device_injection(vd, NULL, ECHILD);
2029         return (required);
2030 }
2032 /*
2033  * Determine if resilver is needed, and if so the txg range.
2034  */
2035 boolean_t
2036 vdev_resilver_needed(vdev_t *vd, uint64_t *minp, uint64_t *maxp)
2037 {
2038         boolean_t needed = B_FALSE;
2039         uint64_t thismin = UINT64_MAX;
2040         uint64_t thismax = 0;
2042         if (vd->vdev_children == 0) {
2043                 mutex_enter(&vd->vdev_dtl_lock);
2044                 if (range_tree_space(vd->vdev_dtl[DTL_MISSING]) != 0 &&
2045                     vdev_writeable(vd)) {
2047                         thismin = vdev_dtl_min(vd);
2048                         thismax = vdev_dtl_max(vd);
2049                         needed = B_TRUE;
2050                 }
2051                 mutex_exit(&vd->vdev_dtl_lock);
2052         } else {
2053                 for (int c = 0; c < vd->vdev_children; c++) {
2054                         vdev_t *cvd = vd->vdev_child[c];
2055                         uint64_t cmin, cmax;
2057                         if (vdev_resilver_needed(cvd, &cmin, &cmax)) {
2058                                 thismin = MIN(thismin, cmin);
2059                                 thismax = MAX(thismax, cmax);
2060                                 needed = B_TRUE;
2061                         }
2062                 }
2063         }
2065         if (needed && minp) {
2066                 *minp = thismin;
2067                 *maxp = thismax;
2068         }
2069         return (needed);
2070 }
2072 void
2073 vdev_load(vdev_t *vd)
2074 {
2075         /*
2076          * Recursively load all children.
2077          */
```



```
2078         for (int c = 0; c < vd->vdev_children; c++)
2079                 vdev_load(vd->vdev_child[c]);
2081         /*
2082          * If this is a top-level vdev, initialize its metaslabs.
2083          */
2084         if (vd == vd->vdev_top && !vd->vdev_ishole &&
2085             (vd->vdev_ashift == 0 || vd->vdev_asize == 0 ||
2086             vdev_metaslab_init(vd, 0) != 0))
2087                 vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2088                     VDEV_AUX_CORRUPT_DATA);
2090         /*
2091          * If this is a leaf vdev, load its DTL.
2092          */
2093         if (vd->vdev_ops->vdev_op_leaf && vdev_dtl_load(vd) != 0)
2094                 vdev_set_state(vd, B_FALSE, VDEV_STATE_CANT_OPEN,
2095                     VDEV_AUX_CORRUPT_DATA);
2096 }
2098 /*
2099  * The special vdev case is used for hot spares and l2cache devices.  Its
2100  * sole purpose it to set the vdev state for the associated vdev.  To do this,
2101  * we make sure that we can open the underlying device, then try to read the
2102  * label, and make sure that the label is sane and that it hasn't been
2103  * repurposed to another pool.
2104  */
2105 int
2106 vdev_validate_aux(vdev_t *vd)
2107 {
2108         nvlist_t *label;
2109         uint64_t guid, version;
2110         uint64_t state;
2112         if (!vdev_readable(vd))
2113                 return (0);
2115         if ((label = vdev_label_read_config(vd, -1ULL)) == NULL) {
2116                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
2117                     VDEV_AUX_CORRUPT_DATA);
2118                 return (-1);
2119         }
2121         if (nvlist_lookup_uint64(label, ZPOOL_CONFIG_VERSION, &version) != 0 ||
2122             !SPA_VERSION_IS_SUPPORTED(version) ||
2123             nvlist_lookup_uint64(label, ZPOOL_CONFIG_GUID, &guid) != 0 ||
2124             guid != vd->vdev_guid ||
2125             nvlist_lookup_uint64(label, ZPOOL_CONFIG_POOL_STATE, &state) != 0) {
2126                 vdev_set_state(vd, B_TRUE, VDEV_STATE_CANT_OPEN,
2127                     VDEV_AUX_CORRUPT_DATA);
2128                 nvlist_free(label);
2129                 return (-1);
2130         }
2132         /*
2133          * We don't actually check the pool state here.  If it's in fact in
2134          * use by another pool, we update this fact on the fly when requested.
2135          */
2136         nvlist_free(label);
2137         return (0);
2138 }
2140 void
2141 vdev_remove(vdev_t *vd, uint64_t txg)
2142 {
2143         spa_t *spa = vd->vdev_spa;
```

```
2144         objset_t *mos = spa->spa_meta_objset;
2145         dmu_tx_t *tx;

2147         tx = dmu_tx_create_assigned(spa_get_dsl(spa), txg);

2149         if (vd->vdev_ms != NULL) {
2150                 metaslab_group_t *mg = vd->vdev_mg;

2152                 metaslab_group_histogram_verify(mg);
2153                 metaslab_class_histogram_verify(mg->mg_class);

2155                 for (int m = 0; m < vd->vdev_ms_count; m++) {
2156                         metaslab_t *msp = vd->vdev_ms[m];

2158                         if (msp == NULL || msp->ms_sm == NULL)
2159                                 continue;

2161                         mutex_enter(&msp->ms_lock);
2162                         /*
2163                          * If the metaslab was not loaded when the vdev
2164                          * was removed then the histogram accounting may
2165                          * not be accurate. Update the histogram information
2166                          * here so that we ensure that the metaslab group
2167                          * and metaslab class are up-to-date.
2168                          */
2169                         metaslab_group_histogram_remove(mg, msp);

2171                         VERIFY0(space_map_allocated(msp->ms_sm));
2172                         space_map_free(msp->ms_sm, tx);
2173                         space_map_close(msp->ms_sm);
2174                         msp->ms_sm = NULL;
2175                         mutex_exit(&msp->ms_lock);
2176                 }

2178                 metaslab_group_histogram_verify(mg);
2179                 metaslab_class_histogram_verify(mg->mg_class);
2180                 for (int i = 0; i < RANGE_TREE_HISTOGRAM_SIZE; i++)
2181                         ASSERT0(mg->mg_histogram[i]);

2183         }

2185         if (vd->vdev_ms_array) {
2186                 (void) dmu_object_free(mos, vd->vdev_ms_array, tx);
2187                 vd->vdev_ms_array = 0;
2188         }
2189         dmu_tx_commit(tx);
2190 }

2192 void
2193 vdev_sync_done(vdev_t *vd, uint64_t txg)
2194 {
2195         metaslab_t *msp;
2196         boolean_t reassess = !txg_list_empty(&vd->vdev_ms_list, TXG_CLEAN(txg));

2198         ASSERT(!vd->vdev_ishole);

2200         while (msp = txg_list_remove(&vd->vdev_ms_list, TXG_CLEAN(txg)))
2201                 metaslab_sync_done(msp, txg);

2203         if (reassess)
2204                 metaslab_sync_reassess(vd->vdev_mg);
2205 }

2207 void
2208 vdev_sync(vdev_t *vd, uint64_t txg)
2209 {
```



```
2210         spa_t *spa = vd->vdev_spa;
2211         vdev_t *lvd;
2212         metaslab_t *msp;
2213         dmu_tx_t *tx;

2215         ASSERT(!vd->vdev_ishole);

2217         if (vd->vdev_ms_array == 0 && vd->vdev_ms_shift != 0) {
2218                 ASSERT(vd == vd->vdev_top);
2219                 tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);
2220                 vd->vdev_ms_array = dmu_object_alloc(spa->spa_meta_objset,
2221                     DMU_OT_OBJECT_ARRAY, 0, DMU_OT_NONE, 0, tx);
2222                 ASSERT(vd->vdev_ms_array != 0);
2223                 vdev_config_dirty(vd);
2224                 dmu_tx_commit(tx);
2225         }

2227         /*
2228          * Remove the metadata associated with this vdev once it's empty.
2229          */
2230         if (vd->vdev_stat.vs_alloc == 0 && vd->vdev_removing)
2231                 vdev_remove(vd, txg);

2233         while ((msp = txg_list_remove(&vd->vdev_ms_list, txg)) != NULL) {
2234                 metaslab_sync(msp, txg);
2235                 (void) txg_list_add(&vd->vdev_ms_list, msp, TXG_CLEAN(txg));
2236         }

2238         while ((lvd = txg_list_remove(&vd->vdev_dtl_list, txg)) != NULL)
2239                 vdev_dtl_sync(lvd, txg);

2241         (void) txg_list_add(&spa->spa_vdev_txg_list, vd, TXG_CLEAN(txg));
2242 }

2244 uint64_t
2245 vdev_psize_to_asize(vdev_t *vd, uint64_t psize)
2246 {
2247         return (vd->vdev_ops->vdev_op_asize(vd, psize));
2248 }

2250 /*
2251  * Mark the given vdev faulted.  A faulted vdev behaves as if the device could
2252  * not be opened, and no I/O is attempted.
2253  */
2254 int
2255 vdev_fault(spa_t *spa, uint64_t guid, vdev_aux_t aux)
2256 {
2257         vdev_t *vd, *tvd;

2259         spa_vdev_state_enter(spa, SCL_NONE);

2261         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2262                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2264         if (!vd->vdev_ops->vdev_op_leaf)
2265                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2267         tvd = vd->vdev_top;

2269         /*
2270          * We don't directly use the aux state here, but if we do a
2271          * vdev_reopen(), we need this value to be present to remember why we
2272          * were faulted.
2273          */
2274         vd->vdev_label_aux = aux;
```

```
2276         /*
2277          * Faulted state takes precedence over degraded.
2278          */
2279         vd->vdev_delayed_close = B_FALSE;
2280         vd->vdev_faulted = 1ULL;
2281         vd->vdev_degraded = 0ULL;
2282         vdev_set_state(vd, B_FALSE, VDEV_STATE_FAULTED, aux);

2284         /*
2285          * If this device has the only valid copy of the data, then
2286          * back off and simply mark the vdev as degraded instead.
2287          */
2288         if (!tvd->vdev_islog && vd->vdev_aux == NULL && vdev_dtl_required(vd)) {
2289                 vd->vdev_degraded = 1ULL;
2290                 vd->vdev_faulted = 0ULL;

2292                 /*
2293                  * If we reopen the device and it's not dead, only then do we
2294                  * mark it degraded.
2295                  */
2296                 vdev_reopen(tvd);

2298                 if (vdev_readable(vd))
2299                         vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED, aux);
2300         }

2302         return (spa_vdev_state_exit(spa, vd, 0));
2303 }

2305 /*
2306  * Mark the given vdev degraded.  A degraded vdev is purely an indication to the
2307  * user that something is wrong.  The vdev continues to operate as normal as far
2308  * as I/O is concerned.
2309  */
2310 int
2311 vdev_degrade(spa_t *spa, uint64_t guid, vdev_aux_t aux)
2312 {
2313         vdev_t *vd;

2315         spa_vdev_state_enter(spa, SCL_NONE);

2317         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2318                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2320         if (!vd->vdev_ops->vdev_op_leaf)
2321                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2323         /*
2324          * If the vdev is already faulted, then don't do anything.
2325          */
2326         if (vd->vdev_faulted || vd->vdev_degraded)
2327                 return (spa_vdev_state_exit(spa, NULL, 0));

2329         vd->vdev_degraded = 1ULL;
2330         if (!vdev_is_dead(vd))
2331                 vdev_set_state(vd, B_FALSE, VDEV_STATE_DEGRADED,
2332                     aux);

2334         return (spa_vdev_state_exit(spa, vd, 0));
2335 }

2337 /*
2338  * Online the given vdev.
2339  *
2340  * If 'ZFS_ONLINE_UNSPARE' is set, it implies two things.  First, any attached
2341  * spare device should be detached when the device finishes resilvering.
```



```
2342  * Second, the online should be treated like a 'test' online case, so no FMA
2343  * events are generated if the device fails to open.
2344  */
2345 int
2346 vdev_online(spa_t *spa, uint64_t guid, uint64_t flags, vdev_state_t *newstate)
2347 {
2348         vdev_t *vd, *tvd, *pvd, *rvd = spa->spa_root_vdev;

2350         spa_vdev_state_enter(spa, SCL_NONE);

2352         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2353                 return (spa_vdev_state_exit(spa, NULL, ENODEV));

2355         if (!vd->vdev_ops->vdev_op_leaf)
2356                 return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2358         tvd = vd->vdev_top;
2359         vd->vdev_offline = B_FALSE;
2360         vd->vdev_tmpoffline = B_FALSE;
2361         vd->vdev_checkremove = !!(flags & ZFS_ONLINE_CHECKREMOVE);
2362         vd->vdev_forcefault = !!(flags & ZFS_ONLINE_FORCEFAULT);

2364         /* XXX - L2ARC 1.0 does not support expansion */
2365         if (!vd->vdev_aux) {
2366                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2367                         pvd->vdev_expanding = !!(flags & ZFS_ONLINE_EXPAND);
2368         }

2370         vdev_reopen(tvd);
2371         vd->vdev_checkremove = vd->vdev_forcefault = B_FALSE;

2373         if (!vd->vdev_aux) {
2374                 for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2375                         pvd->vdev_expanding = B_FALSE;
2376         }

2378         if (newstate)
2379                 *newstate = vd->vdev_state;
2380         if ((flags & ZFS_ONLINE_UNSPARE) &&
2381             !vdev_is_dead(vd) && vd->vdev_parent &&
2382             vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
2383             vd->vdev_parent->vdev_child[0] == vd)
2384                 vd->vdev_unspare = B_TRUE;

2386         if ((flags & ZFS_ONLINE_EXPAND) || spa->spa_autoexpand) {

2388                 /* XXX - L2ARC 1.0 does not support expansion */
2389                 if (vd->vdev_aux)
2390                         return (spa_vdev_state_exit(spa, vd, ENOTSUP));
2391                 spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);
2392         }
2393         return (spa_vdev_state_exit(spa, vd, 0));
2394 }

2396 static int
2397 vdev_offline_locked(spa_t *spa, uint64_t guid, uint64_t flags)
2398 {
2399         vdev_t *vd, *tvd;
2400         int error = 0;
2401         uint64_t generation;
2402         metaslab_group_t *mg;

2404 top:
2405         spa_vdev_state_enter(spa, SCL_ALLOC);

2407         if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
```

```
2408                return (spa_vdev_state_exit(spa, NULL, ENODEV));

2410        if (!vd->vdev_ops->vdev_op_leaf)
2411                return (spa_vdev_state_exit(spa, NULL, ENOTSUP));

2413        tvd = vd->vdev_top;
2414        mg = tvd->vdev_mg;
2415        generation = spa->spa_config_generation + 1;

2417        /*
2418         * If the device isn't already offline, try to offline it.
2419         */
2420        if (!vd->vdev_offline) {
2421                /*
2422                 * If this device has the only valid copy of some data,
2423                 * don't allow it to be offlined. Log devices are always
2424                 * expendable.
2425                 */
2426                if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2427                    vdev_dtl_required(vd))
2428                        return (spa_vdev_state_exit(spa, NULL, EBUSY));

2430                /*
2431                 * If the top-level is a slog and it has had allocations
2432                 * then proceed.  We check that the vdev's metaslab group
2433                 * is not NULL since it's possible that we may have just
2434                 * added this vdev but not yet initialized its metaslabs.
2435                 */
2436                if (tvd->vdev_islog && mg != NULL) {
2437                        /*
2438                         * Prevent any future allocations.
2439                         */
2440                        metaslab_group_passivate(mg);
2441                        (void) spa_vdev_state_exit(spa, vd, 0);

2443                        error = spa_offline_log(spa);

2445                        spa_vdev_state_enter(spa, SCL_ALLOC);

2447                        /*
2448                         * Check to see if the config has changed.
2449                         */
2450                        if (error || generation != spa->spa_config_generation) {
2451                                metaslab_group_activate(mg);
2452                                if (error)
2453                                        return (spa_vdev_state_exit(spa,
2454                                            vd, error));
2455                                (void) spa_vdev_state_exit(spa, vd, 0);
2456                                goto top;
2457                        }
2458                        ASSERT0(tvd->vdev_stat.vs_alloc);
2459                }

2461                /*
2462                 * Offline this device and reopen its top-level vdev.
2463                 * If the top-level vdev is a log device then just offline
2464                 * it. Otherwise, if this action results in the top-level
2465                 * vdev becoming unusable, undo it and fail the request.
2466                 */
2467                vd->vdev_offline = B_TRUE;
2468                vdev_reopen(tvd);

2470                if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2471                    vdev_is_dead(tvd)) {
2472                        vd->vdev_offline = B_FALSE;
2473                        vdev_reopen(tvd);
```



```
2474                        return (spa_vdev_state_exit(spa, NULL, EBUSY));
2475                }

2477                /*
2478                 * Add the device back into the metaslab rotor so that
2479                 * once we online the device it's open for business.
2480                 */
2481                if (tvd->vdev_islog && mg != NULL)
2482                        metaslab_group_activate(mg);
2483        }

2485        vd->vdev_tmpoffline = !!(flags & ZFS_OFFLINE_TEMPORARY);

2487        return (spa_vdev_state_exit(spa, vd, 0));
2488 }

2490 int
2491 vdev_offline(spa_t *spa, uint64_t guid, uint64_t flags)
2492 {
2493        int error;

2495        mutex_enter(&spa->spa_vdev_top_lock);
2496        error = vdev_offline_locked(spa, guid, flags);
2497        mutex_exit(&spa->spa_vdev_top_lock);

2499        return (error);
2500 }

2502 /*
2503  * Clear the error counts associated with this vdev.  Unlike vdev_online() and
2504  * vdev_offline(), we assume the spa config is locked.  We also clear all
2505  * children.  If 'vd' is NULL, then the user wants to clear all vdevs.
2506  */
2507 void
2508 vdev_clear(spa_t *spa, vdev_t *vd)
2509 {
2510        vdev_t *rvd = spa->spa_root_vdev;

2512        ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);

2514        if (vd == NULL)
2515                vd = rvd;

2517        vd->vdev_stat.vs_read_errors = 0;
2518        vd->vdev_stat.vs_write_errors = 0;
2519        vd->vdev_stat.vs_checksum_errors = 0;

2521        for (int c = 0; c < vd->vdev_children; c++)
2522                vdev_clear(spa, vd->vdev_child[c]);

2524        /*
2525         * If we're in the FAULTED state or have experienced failed I/O, then
2526         * clear the persistent state and attempt to reopen the device.  We
2527         * also mark the vdev config dirty, so that the new faulted state is
2528         * written out to disk.
2529         */
2530        if (vd->vdev_faulted || vd->vdev_degraded ||
2531            !vdev_readable(vd) || !vdev_writeable(vd)) {

2533                /*
2534                 * When reopening in reponse to a clear event, it may be due to
2535                 * a fmadm repair request.  In this case, if the device is
2536                 * still broken, we want to still post the ereport again.
2537                 */
2538                vd->vdev_forcefault = B_TRUE;
```

```
2540                 vd->vdev_faulted = vd->vdev_degraded = 0ULL;
2541                 vd->vdev_cant_read = B_FALSE;
2542                 vd->vdev_cant_write = B_FALSE;

2544                 vdev_reopen(vd == rvd ? rvd : vd->vdev_top);

2546                 vd->vdev_forcefault = B_FALSE;

2548                 if (vd != rvd && vdev_writeable(vd->vdev_top))
2549                         vdev_state_dirty(vd->vdev_top);

2551                 if (vd->vdev_aux == NULL && !vdev_is_dead(vd))
2552                         spa_async_request(spa, SPA_ASYNC_RESILVER);

2554                 spa_event_notify(spa, vd, ESC_ZFS_VDEV_CLEAR);
2555         }

2557         /*
2558          * When clearing a FMA-diagnosed fault, we always want to
2559          * unspare the device, as we assume that the original spare was
2560          * done in response to the FMA fault.
2561          */
2562         if (!vdev_is_dead(vd) && vd->vdev_parent != NULL &&
2563             vd->vdev_parent->vdev_ops == &vdev_spare_ops &&
2564             vd->vdev_parent->vdev_child[0] == vd)
2565                 vd->vdev_unspare = B_TRUE;
2566 }

2568 boolean_t
2569 vdev_is_dead(vdev_t *vd)
2570 {
2571         /*
2572          * Holes and missing devices are always considered "dead".
2573          * This simplifies the code since we don't have to check for
2574          * these types of devices in the various code paths.
2575          * Instead we rely on the fact that we skip over dead devices
2576          * before issuing I/O to them.
2577          */
2578         return (vd->vdev_state < VDEV_STATE_DEGRADED || vd->vdev_ishole ||
2579             vd->vdev_ops == &vdev_missing_ops);
2580 }

2582 boolean_t
2583 vdev_readable(vdev_t *vd)
2584 {
2585         return (!vdev_is_dead(vd) && !vd->vdev_cant_read);
2586 }

2588 boolean_t
2589 vdev_writeable(vdev_t *vd)
2590 {
2591         return (!vdev_is_dead(vd) && !vd->vdev_cant_write);
2592 }

2594 boolean_t
2595 vdev_allocatable(vdev_t *vd)
2596 {
2597         uint64_t state = vd->vdev_state;

2599         /*
2600          * We currently allow allocations from vdevs which may be in the
2601          * process of reopening (i.e. VDEV_STATE_CLOSED). If the device
2602          * fails to reopen then we'll catch it later when we're holding
2603          * the proper locks.  Note that we have to get the vdev state
2604          * in a local variable because although it changes atomically,
2605          * we're asking two separate questions about it.
```



```
2606          */
2607         return (!(state < VDEV_STATE_DEGRADED && state != VDEV_STATE_CLOSED) &&
2608             !vd->vdev_cant_write && !vd->vdev_ishole);
2609 }

2611 boolean_t
2612 vdev_accessible(vdev_t *vd, zio_t *zio)
2613 {
2614         ASSERT(zio->io_vd == vd);

2616         if (vdev_is_dead(vd) || vd->vdev_remove_wanted)
2617                 return (B_FALSE);

2619         if (zio->io_type == ZIO_TYPE_READ)
2620                 return (!vd->vdev_cant_read);

2622         if (zio->io_type == ZIO_TYPE_WRITE)
2623                 return (!vd->vdev_cant_write);

2625         return (B_TRUE);
2626 }

2628 /*
2629  * Get statistics for the given vdev.
2630  */
2631 void
2632 vdev_get_stats(vdev_t *vd, vdev_stat_t *vs)
2633 {
2634         spa_t *spa = vd->vdev_spa;
2635         vdev_t *rvd = spa->spa_root_vdev;

2637         ASSERT(spa_config_held(spa, SCL_ALL, RW_READER) != 0);

2639         mutex_enter(&vd->vdev_stat_lock);
2640         bcopy(&vd->vdev_stat, vs, sizeof (*vs));
2641         vs->vs_timestamp = gethrtime() - vs->vs_timestamp;
2642         vs->vs_state = vd->vdev_state;
2643         vs->vs_rsize = vdev_get_min_asize(vd);
2644         if (vd->vdev_ops->vdev_op_leaf)
2645                 vs->vs_rsize += VDEV_LABEL_START_SIZE + VDEV_LABEL_END_SIZE;
2646         vs->vs_esize = vd->vdev_max_asize - vd->vdev_asize;
2647         if (vd->vdev_aux == NULL && vd == vd->vdev_top && !vd->vdev_ishole) {
2648                 vs->vs_fragmentation = vd->vdev_mg->mg_fragmentation;
2649         }

2651         /*
2652          * If we're getting stats on the root vdev, aggregate the I/O counts
2653          * over all top-level vdevs (i.e. the direct children of the root).
2654          */
2655         if (vd == rvd) {
2656                 for (int c = 0; c < rvd->vdev_children; c++) {
2657                         vdev_t *cvd = rvd->vdev_child[c];
2658                         vdev_stat_t *cvs = &cvd->vdev_stat;

2660                         for (int t = 0; t < ZIO_TYPES; t++) {
2661                                 vs->vs_ops[t] += cvs->vs_ops[t];
2662                                 vs->vs_bytes[t] += cvs->vs_bytes[t];
2663                         }
2664                         cvs->vs_scan_removing = cvd->vdev_removing;
2665                 }
2666         }
2667         mutex_exit(&vd->vdev_stat_lock);
2668 }

2670 void
2671 vdev_clear_stats(vdev_t *vd)
```

```
2672 {
2673         mutex_enter(&vd->vdev_stat_lock);
2674         vd->vdev_stat.vs_space = 0;
2675         vd->vdev_stat.vs_dspace = 0;
2676         vd->vdev_stat.vs_alloc = 0;
2677         mutex_exit(&vd->vdev_stat_lock);
2678 }

2680 void
2681 vdev_scan_stat_init(vdev_t *vd)
2682 {
2683         vdev_stat_t *vs = &vd->vdev_stat;

2685         for (int c = 0; c < vd->vdev_children; c++)
2686                 vdev_scan_stat_init(vd->vdev_child[c]);

2688         mutex_enter(&vd->vdev_stat_lock);
2689         vs->vs_scan_processed = 0;
2690         mutex_exit(&vd->vdev_stat_lock);
2691 }

2693 void
2694 vdev_stat_update(zio_t *zio, uint64_t psize)
2695 {
2696         spa_t *spa = zio->io_spa;
2697         vdev_t *rvd = spa->spa_root_vdev;
2698         vdev_t *vd = zio->io_vd ? zio->io_vd : rvd;
2699         vdev_t *pvd;
2700         uint64_t txg = zio->io_txg;
2701         vdev_stat_t *vs = &vd->vdev_stat;
2702         zio_type_t type = zio->io_type;
2703         int flags = zio->io_flags;

2705         /*
2706          * If this i/o is a gang leader, it didn't do any actual work.
2707          */
2708         if (zio->io_gang_tree)
2709                 return;

2711         if (zio->io_error == 0) {
2712                 /*
2713                  * If this is a root i/o, don't count it -- we've already
2714                  * counted the top-level vdevs, and vdev_get_stats() will
2715                  * aggregate them when asked.  This reduces contention on
2716                  * the root vdev_stat_lock and implicitly handles blocks
2717                  * that compress away to holes, for which there is no i/o.
2718                  * (Holes never create vdev children, so all the counters
2719                  * remain zero, which is what we want.)
2720                  *
2721                  * Note: this only applies to successful i/o (io_error == 0)
2722                  * because unlike i/o counts, errors are not additive.
2723                  * When reading a ditto block, for example, failure of
2724                  * one top-level vdev does not imply a root-level error.
2725                  */
2726                 if (vd == rvd)
2727                         return;

2729                 ASSERT(vd == zio->io_vd);

2731                 if (flags & ZIO_FLAG_IO_BYPASS)
2732                         return;

2734                 mutex_enter(&vd->vdev_stat_lock);

2736                 if (flags & ZIO_FLAG_IO_REPAIR) {
2737                         if (flags & ZIO_FLAG_SCAN_THREAD) {
```



```
2738                                 dsl_scan_phys_t *scn_phys =
2739                                     &spa->spa_dsl_pool->dp_scan->scn_phys;
2740                                 uint64_t *processed = &scn_phys->scn_processed;

2742                                 /* XXX cleanup? */
2743                                 if (vd->vdev_ops->vdev_op_leaf)
2744                                         atomic_add_64(processed, psize);
2745                                 vs->vs_scan_processed += psize;
2746                         }

2748                         if (flags & ZIO_FLAG_SELF_HEAL)
2749                                 vs->vs_self_healed += psize;
2750                 }

2752                 vs->vs_ops[type]++;
2753                 vs->vs_bytes[type] += psize;

2755                 mutex_exit(&vd->vdev_stat_lock);
2756                 return;
2757         }

2759         if (flags & ZIO_FLAG_SPECULATIVE)
2760                 return;

2762         /*
2763          * If this is an I/O error that is going to be retried, then ignore the
2764          * error.  Otherwise, the user may interpret B_FAILFAST I/O errors as
2765          * hard errors, when in reality they can happen for any number of
2766          * innocuous reasons (bus resets, MPxIO link failure, etc).
2767          */
2768         if (zio->io_error == EIO &&
2769             !(zio->io_flags & ZIO_FLAG_IO_RETRY))
2770                 return;

2772         /*
2773          * Intent logs writes won't propagate their error to the root
2774          * I/O so don't mark these types of failures as pool-level
2775          * errors.
2776          */
2777         if (zio->io_vd == NULL && (zio->io_flags & ZIO_FLAG_DONT_PROPAGATE))
2778                 return;

2780         mutex_enter(&vd->vdev_stat_lock);
2781         if (type == ZIO_TYPE_READ && !vdev_is_dead(vd)) {
2782                 if (zio->io_error == ECKSUM)
2783                         vs->vs_checksum_errors++;
2784                 else
2785                         vs->vs_read_errors++;
2786         }
2787         if (type == ZIO_TYPE_WRITE && !vdev_is_dead(vd))
2788                 vs->vs_write_errors++;
2789         mutex_exit(&vd->vdev_stat_lock);

2791         if (type == ZIO_TYPE_WRITE && txg != 0 &&
2792             (!(flags & ZIO_FLAG_IO_REPAIR) ||
2793             (flags & ZIO_FLAG_SCAN_THREAD) ||
2794             spa->spa_claiming)) {
2795                 /*
2796                  * This is either a normal write (not a repair), or it's
2797                  * a repair induced by the scrub thread, or it's a repair
2798                  * made by zil_claim() during spa_load() in the first txg.
2799                  * In the normal case, we commit the DTL change in the same
2800                  * txg as the block was born.  In the scrub-induced repair
2801                  * case, we know that scrubs run in first-pass syncing context,
2802                  * so we commit the DTL change in spa_syncing_txg(spa).
2803                  * In the zil_claim() case, we commit in spa_first_txg(spa).
```

```
2804                  *
2805                  * We currently do not make DTL entries for failed spontaneous
2806                  * self-healing writes triggered by normal (non-scrubbing)
2807                  * reads, because we have no transactional context in which to
2808                  * do so -- and it's not clear that it'd be desirable anyway.
2809                  */
2810                 if (vd->vdev_ops->vdev_op_leaf) {
2811                         uint64_t commit_txg = txg;
2812                         if (flags & ZIO_FLAG_SCAN_THREAD) {
2813                                 ASSERT(flags & ZIO_FLAG_IO_REPAIR);
2814                                 ASSERT(spa_sync_pass(spa) == 1);
2815                                 vdev_dtl_dirty(vd, DTL_SCRUB, txg, 1);
2816                                 commit_txg = spa_syncing_txg(spa);
2817                         } else if (spa->spa_claiming) {
2818                                 ASSERT(flags & ZIO_FLAG_IO_REPAIR);
2819                                 commit_txg = spa_first_txg(spa);
2820                         }
2821                         ASSERT(commit_txg >= spa_syncing_txg(spa));
2822                         if (vdev_dtl_contains(vd, DTL_MISSING, txg, 1))
2823                                 return;
2824                         for (pvd = vd; pvd != rvd; pvd = pvd->vdev_parent)
2825                                 vdev_dtl_dirty(pvd, DTL_PARTIAL, txg, 1);
2826                         vdev_dirty(vd->vdev_top, VDD_DTL, vd, commit_txg);
2827                 }
2828                 if (vd != rvd)
2829                         vdev_dtl_dirty(vd, DTL_MISSING, txg, 1);
2830         }
2831 }

2833 /*
2834  * Update the in-core space usage stats for this vdev, its metaslab class,
2835  * and the root vdev.
2836  */
2837 void
2838 vdev_space_update(vdev_t *vd, int64_t alloc_delta, int64_t defer_delta,
2839     int64_t space_delta)
2840 {
2841         int64_t dspace_delta = space_delta;
2842         spa_t *spa = vd->vdev_spa;
2843         vdev_t *rvd = spa->spa_root_vdev;
2844         metaslab_group_t *mg = vd->vdev_mg;
2845         metaslab_class_t *mc = mg ? mg->mg_class : NULL;

2847         ASSERT(vd == vd->vdev_top);

2849         /*
2850          * Apply the inverse of the psize-to-asize (ie. RAID-Z) space-expansion
2851          * factor.  We must calculate this here and not at the root vdev
2852          * because the root vdev's psize-to-asize is simply the max of its
2853          * childrens', thus not accurate enough for us.
2854          */
2855         ASSERT((dspace_delta & (SPA_MINBLOCKSIZE-1)) == 0);
2856         ASSERT(vd->vdev_deflate_ratio != 0 || vd->vdev_isl2cache);
2857         dspace_delta = (dspace_delta >> SPA_MINBLOCKSHIFT) *
2858             vd->vdev_deflate_ratio;

2860         mutex_enter(&vd->vdev_stat_lock);
2861         vd->vdev_stat.vs_alloc += alloc_delta;
2862         vd->vdev_stat.vs_space += space_delta;
2863         vd->vdev_stat.vs_dspace += dspace_delta;
2864         mutex_exit(&vd->vdev_stat_lock);

2866         if (mc == spa_normal_class(spa)) {
2867                 mutex_enter(&rvd->vdev_stat_lock);
2868                 rvd->vdev_stat.vs_alloc += alloc_delta;
2869                 rvd->vdev_stat.vs_space += space_delta;
```



```
2870                 rvd->vdev_stat.vs_dspace += dspace_delta;
2871                 mutex_exit(&rvd->vdev_stat_lock);
2872         }

2874         if (mc != NULL) {
2875                 ASSERT(rvd == vd->vdev_parent);
2876                 ASSERT(vd->vdev_ms_count != 0);

2878                 metaslab_class_space_update(mc,
2879                     alloc_delta, defer_delta, space_delta, dspace_delta);
2880         }
2881 }

2883 /*
2884  * Mark a top-level vdev's config as dirty, placing it on the dirty list
2885  * so that it will be written out next time the vdev configuration is synced.
2886  * If the root vdev is specified (vdev_top == NULL), dirty all top-level vdevs.
2887  */
2888 void
2889 vdev_config_dirty(vdev_t *vd)
2890 {
2891         spa_t *spa = vd->vdev_spa;
2892         vdev_t *rvd = spa->spa_root_vdev;
2893         int c;

2895         ASSERT(spa_writeable(spa));

2897         /*
2898          * If this is an aux vdev (as with l2cache and spare devices), then we
2899          * update the vdev config manually and set the sync flag.
2900          */
2901         if (vd->vdev_aux != NULL) {
2902                 spa_aux_vdev_t *sav = vd->vdev_aux;
2903                 nvlist_t **aux;
2904                 uint_t naux;

2906                 for (c = 0; c < sav->sav_count; c++) {
2907                         if (sav->sav_vdevs[c] == vd)
2908                                 break;
2909                 }

2911                 if (c == sav->sav_count) {
2912                         /*
2913                          * We're being removed.  There's nothing more to do.
2914                          */
2915                         ASSERT(sav->sav_sync == B_TRUE);
2916                         return;
2917                 }

2919                 sav->sav_sync = B_TRUE;

2921                 if (nvlist_lookup_nvlist_array(sav->sav_config,
2922                     ZPOOL_CONFIG_L2CACHE, &aux, &naux) != 0) {
2923                         VERIFY(nvlist_lookup_nvlist_array(sav->sav_config,
2924                             ZPOOL_CONFIG_SPARES, &aux, &naux) == 0);
2925                 }

2927                 ASSERT(c < naux);

2929                 /*
2930                  * Setting the nvlist in the middle if the array is a little
2931                  * sketchy, but it will work.
2932                  */
2933                 nvlist_free(aux[c]);
2934                 aux[c] = vdev_config_generate(spa, vd, B_TRUE, 0);
```

```
2936                 return;
2937         }
2938
2939         /*
2940          * The dirty list is protected by the SCL_CONFIG lock.  The caller
2941          * must either hold SCL_CONFIG as writer, or must be the sync thread
2942          * (which holds SCL_CONFIG as reader).  There's only one sync thread,
2943          * so this is sufficient to ensure mutual exclusion.
2944          */
2945         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
2946             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2947             spa_config_held(spa, SCL_CONFIG, RW_READER)));
2948
2949         if (vd == rvd) {
2950                 for (c = 0; c < rvd->vdev_children; c++)
2951                         vdev_config_dirty(rvd->vdev_child[c]);
2952         } else {
2953                 ASSERT(vd == vd->vdev_top);
2954
2955                 if (!list_link_active(&vd->vdev_config_dirty_node) &&
2956                     !vd->vdev_ishole)
2957                         list_insert_head(&spa->spa_config_dirty_list, vd);
2958         }
2959 }
2960
2961 void
2962 vdev_config_clean(vdev_t *vd)
2963 {
2964         spa_t *spa = vd->vdev_spa;
2965
2966         ASSERT(spa_config_held(spa, SCL_CONFIG, RW_WRITER) ||
2967             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2968             spa_config_held(spa, SCL_CONFIG, RW_READER)));
2969
2970         ASSERT(list_link_active(&vd->vdev_config_dirty_node));
2971         list_remove(&spa->spa_config_dirty_list, vd);
2972 }
2973
2974 /*
2975  * Mark a top-level vdev's state as dirty, so that the next pass of
2976  * spa_sync() can convert this into vdev_config_dirty().  We distinguish
2977  * the state changes from larger config changes because they require
2978  * much less locking, and are often needed for administrative actions.
2979  */
2980 void
2981 vdev_state_dirty(vdev_t *vd)
2982 {
2983         spa_t *spa = vd->vdev_spa;
2984
2985         ASSERT(spa_writeable(spa));
2986         ASSERT(vd == vd->vdev_top);
2987
2988         /*
2989          * The state list is protected by the SCL_STATE lock.  The caller
2990          * must either hold SCL_STATE as writer, or must be the sync thread
2991          * (which holds SCL_STATE as reader).  There's only one sync thread,
2992          * so this is sufficient to ensure mutual exclusion.
2993          */
2994         ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
2995             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
2996             spa_config_held(spa, SCL_STATE, RW_READER)));
2997
2998         if (!list_link_active(&vd->vdev_state_dirty_node) && !vd->vdev_ishole)
2999                 list_insert_head(&spa->spa_state_dirty_list, vd);
3000 }
```



```
3002 void
3003 vdev_state_clean(vdev_t *vd)
3004 {
3005         spa_t *spa = vd->vdev_spa;
3006
3007         ASSERT(spa_config_held(spa, SCL_STATE, RW_WRITER) ||
3008             (dsl_pool_sync_context(spa_get_dsl(spa)) &&
3009             spa_config_held(spa, SCL_STATE, RW_READER)));
3010
3011         ASSERT(list_link_active(&vd->vdev_state_dirty_node));
3012         list_remove(&spa->spa_state_dirty_list, vd);
3013 }
3014
3015 /*
3016  * Propagate vdev state up from children to parent.
3017  */
3018 void
3019 vdev_propagate_state(vdev_t *vd)
3020 {
3021         spa_t *spa = vd->vdev_spa;
3022         vdev_t *rvd = spa->spa_root_vdev;
3023         int degraded = 0, faulted = 0;
3024         int corrupted = 0;
3025         vdev_t *child;
3026
3027         if (vd->vdev_children > 0) {
3028                 for (int c = 0; c < vd->vdev_children; c++) {
3029                         child = vd->vdev_child[c];
3030
3031                         /*
3032                          * Don't factor holes into the decision.
3033                          */
3034                         if (child->vdev_ishole)
3035                                 continue;
3036
3037                         if (!vdev_readable(child) ||
3038                             (!vdev_writeable(child) && spa_writeable(spa))) {
3039                                 /*
3040                                  * Root special: if there is a top-level log
3041                                  * device, treat the root vdev as if it were
3042                                  * degraded.
3043                                  */
3044                                 if (child->vdev_islog && vd == rvd)
3045                                         degraded++;
3046                                 else
3047                                         faulted++;
3048                         } else if (child->vdev_state <= VDEV_STATE_DEGRADED) {
3049                                 degraded++;
3050                         }
3051
3052                         if (child->vdev_stat.vs_aux == VDEV_AUX_CORRUPT_DATA)
3053                                 corrupted++;
3054                 }
3055
3056                 vd->vdev_ops->vdev_op_state_change(vd, faulted, degraded);
3057
3058                 /*
3059                  * Root special: if there is a top-level vdev that cannot be
3060                  * opened due to corrupted metadata, then propagate the root
3061                  * vdev's aux state as 'corrupt' rather than 'insufficient
3062                  * replicas'.
3063                  */
3064                 if (corrupted && vd == rvd &&
3065                     rvd->vdev_state == VDEV_STATE_CANT_OPEN)
3066                         vdev_set_state(rvd, B_FALSE, VDEV_STATE_CANT_OPEN,
3067                             VDEV_AUX_CORRUPT_DATA);
```

```
3068         }
3069
3070         if (vd->vdev_parent)
3071                 vdev_propagate_state(vd->vdev_parent);
3072 }
3073
3074 /*
3075  * Set a vdev's state.  If this is during an open, we don't update the parent
3076  * state, because we're in the process of opening children depth-first.
3077  * Otherwise, we propagate the change to the parent.
3078  *
3079  * If this routine places a device in a faulted state, an appropriate ereport is
3080  * generated.
3081  */
3082 void
3083 vdev_set_state(vdev_t *vd, boolean_t isopen, vdev_state_t state, vdev_aux_t aux)
3084 {
3085         uint64_t save_state;
3086         spa_t *spa = vd->vdev_spa;
3087
3088         if (state == vd->vdev_state) {
3089                 vd->vdev_stat.vs_aux = aux;
3090                 return;
3091         }
3092
3093         save_state = vd->vdev_state;
3094
3095         vd->vdev_state = state;
3096         vd->vdev_stat.vs_aux = aux;
3097
3098         /*
3099          * If we are setting the vdev state to anything but an open state, then
3100          * always close the underlying device unless the device has requested
3101          * a delayed close (i.e. we're about to remove or fault the device).
3102          * Otherwise, we keep accessible but invalid devices open forever.
3103          * We don't call vdev_close() itself, because that implies some extra
3104          * checks (offline, etc) that we don't want here.  This is limited to
3105          * leaf devices, because otherwise closing the device will affect other
3106          * children.
3107          */
3108         if (!vd->vdev_delayed_close && vdev_is_dead(vd) &&
3109             vd->vdev_ops->vdev_op_leaf)
3110                 vd->vdev_ops->vdev_op_close(vd);
3111
3112         /*
3113          * If we have brought this vdev back into service, we need
3114          * to notify fmd so that it can gracefully repair any outstanding
3115          * cases due to a missing device.  We do this in all cases, even those
3116          * that probably don't correlate to a repaired fault.  This is sure to
3117          * catch all cases, and we let the zfs-retire agent sort it out.  If
3118          * this is a transient state it's OK, as the retire agent will
3119          * double-check the state of the vdev before repairing it.
3120          */
3121         if (state == VDEV_STATE_HEALTHY && vd->vdev_ops->vdev_op_leaf &&
3122             vd->vdev_prevstate != state)
3123                 zfs_post_state_change(spa, vd);
3124
3125         if (vd->vdev_removed &&
3126             state == VDEV_STATE_CANT_OPEN &&
3127             (aux == VDEV_AUX_OPEN_FAILED || vd->vdev_checkremove)) {
3128                 /*
3129                  * If the previous state is set to VDEV_STATE_REMOVED, then this
3130                  * device was previously marked removed and someone attempted to
3131                  * reopen it.  If this failed due to a nonexistent device, then
3132                  * keep the device in the REMOVED state.  We also let this be if
3133                  * it is one of our special test online cases, which is only
```



```
3134                  * attempting to online the device and shouldn't generate an FMA
3135                  * fault.
3136                  */
3137                 vd->vdev_state = VDEV_STATE_REMOVED;
3138                 vd->vdev_stat.vs_aux = VDEV_AUX_NONE;
3139         } else if (state == VDEV_STATE_REMOVED) {
3140                 vd->vdev_removed = B_TRUE;
3141         } else if (state == VDEV_STATE_CANT_OPEN) {
3142                 /*
3143                  * If we fail to open a vdev during an import or recovery, we
3144                  * mark it as "not available", which signifies that it was
3145                  * never there to begin with.  Failure to open such a device
3146                  * is not considered an error.
3147                  */
3148                 if ((spa_load_state(spa) == SPA_LOAD_IMPORT ||
3149                     spa_load_state(spa) == SPA_LOAD_RECOVER) &&
3150                     vd->vdev_ops->vdev_op_leaf)
3151                         vd->vdev_not_present = 1;
3152
3153                 /*
3154                  * Post the appropriate ereport.  If the 'prevstate' field is
3155                  * set to something other than VDEV_STATE_UNKNOWN, it indicates
3156                  * that this is part of a vdev_reopen().  In this case, we don't
3157                  * want to post the ereport if the device was already in the
3158                  * CANT_OPEN state beforehand.
3159                  *
3160                  * If the 'checkremove' flag is set, then this is an attempt to
3161                  * online the device in response to an insertion event.  If we
3162                  * hit this case, then we have detected an insertion event for a
3163                  * faulted or offline device that wasn't in the removed state.
3164                  * In this scenario, we don't post an ereport because we are
3165                  * about to replace the device, or attempt an online with
3166                  * vdev_forcefault, which will generate the fault for us.
3167                  */
3168                 if ((vd->vdev_prevstate != state || vd->vdev_forcefault) &&
3169                     !vd->vdev_not_present && !vd->vdev_checkremove &&
3170                     vd != spa->spa_root_vdev) {
3171                         const char *class;
3172
3173                         switch (aux) {
3174                         case VDEV_AUX_OPEN_FAILED:
3175                                 class = FM_EREPORT_ZFS_DEVICE_OPEN_FAILED;
3176                                 break;
3177                         case VDEV_AUX_CORRUPT_DATA:
3178                                 class = FM_EREPORT_ZFS_DEVICE_CORRUPT_DATA;
3179                                 break;
3180                         case VDEV_AUX_NO_REPLICAS:
3181                                 class = FM_EREPORT_ZFS_DEVICE_NO_REPLICAS;
3182                                 break;
3183                         case VDEV_AUX_BAD_GUID_SUM:
3184                                 class = FM_EREPORT_ZFS_DEVICE_BAD_GUID_SUM;
3185                                 break;
3186                         case VDEV_AUX_TOO_SMALL:
3187                                 class = FM_EREPORT_ZFS_DEVICE_TOO_SMALL;
3188                                 break;
3189                         case VDEV_AUX_BAD_LABEL:
3190                                 class = FM_EREPORT_ZFS_DEVICE_BAD_LABEL;
3191                                 break;
3192                         default:
3193                                 class = FM_EREPORT_ZFS_DEVICE_UNKNOWN;
3194                         }
3195
3196                         zfs_ereport_post(class, spa, vd, NULL, save_state, 0);
3197                 }
3198
3199                 /* Erase any notion of persistent removed state */
```

```
3200                 vd->vdev_removed = B_FALSE;
3201         } else {
3202                 vd->vdev_removed = B_FALSE;
3203         }

3205         if (!isopen && vd->vdev_parent)
3206                 vdev_propagate_state(vd->vdev_parent);
3207 }

3209 /*
3210  * Check the vdev configuration to ensure that it's capable of supporting
3211  * a root pool. Currently, we do not support RAID-Z or partial configuration.
3212  * In addition, only a single top-level vdev is allowed and none of the leaves
3213  * can be wholedisks.
3214  */
3215 boolean_t
3216 vdev_is_bootable(vdev_t *vd)
3217 {
3218         if (!vd->vdev_ops->vdev_op_leaf) {
3219                 char *vdev_type = vd->vdev_ops->vdev_op_type;

3221                 if (strcmp(vdev_type, VDEV_TYPE_ROOT) == 0 &&
3222                     vd->vdev_children > 1) {
3223                         return (B_FALSE);
3224                 } else if (strcmp(vdev_type, VDEV_TYPE_RAIDZ) == 0 ||
3225                     strcmp(vdev_type, VDEV_TYPE_MISSING) == 0) {
3226                         return (B_FALSE);
3227                 }
3228         }

3230         for (int c = 0; c < vd->vdev_children; c++) {
3231                 if (!vdev_is_bootable(vd->vdev_child[c]))
3232                         return (B_FALSE);
3233         }
3234         return (B_TRUE);
3235 }

3237 /*
3238  * Load the state from the original vdev tree (ovd) which
3239  * we've retrieved from the MOS config object. If the original
3240  * vdev was offline or faulted then we transfer that state to the
3241  * device in the current vdev tree (nvd).
3242  */
3243 void
3244 vdev_load_log_state(vdev_t *nvd, vdev_t *ovd)
3245 {
3246         spa_t *spa = nvd->vdev_spa;

3248         ASSERT(nvd->vdev_top->vdev_islog);
3249         ASSERT(spa_config_held(spa, SCL_STATE_ALL, RW_WRITER) == SCL_STATE_ALL);
3250         ASSERT3U(nvd->vdev_guid, ==, ovd->vdev_guid);

3252         for (int c = 0; c < nvd->vdev_children; c++)
3253                 vdev_load_log_state(nvd->vdev_child[c], ovd->vdev_child[c]);

3255         if (nvd->vdev_ops->vdev_op_leaf) {
3256                 /*
3257                  * Restore the persistent vdev state
3258                  */
3259                 nvd->vdev_offline = ovd->vdev_offline;
3260                 nvd->vdev_faulted = ovd->vdev_faulted;
3261                 nvd->vdev_degraded = ovd->vdev_degraded;
3262                 nvd->vdev_removed = ovd->vdev_removed;
3263         }
3264 }
```



```
3266 /*
3267  * Determine if a log device has valid content.  If the vdev was
3268  * removed or faulted in the MOS config then we know that
3269  * the content on the log device has already been written to the pool.
3270  */
3271 boolean_t
3272 vdev_log_state_valid(vdev_t *vd)
3273 {
3274         if (vd->vdev_ops->vdev_op_leaf && !vd->vdev_faulted &&
3275             !vd->vdev_removed)
3276                 return (B_TRUE);

3278         for (int c = 0; c < vd->vdev_children; c++)
3279                 if (vdev_log_state_valid(vd->vdev_child[c]))
3280                         return (B_TRUE);

3282         return (B_FALSE);
3283 }

3285 /*
3286  * Expand a vdev if possible.
3287  */
3288 void
3289 vdev_expand(vdev_t *vd, uint64_t txg)
3290 {
3291         ASSERT(vd->vdev_top == vd);
3292         ASSERT(spa_config_held(vd->vdev_spa, SCL_ALL, RW_WRITER) == SCL_ALL);

3294         if ((vd->vdev_asize >> vd->vdev_ms_shift) > vd->vdev_ms_count) {
3295                 VERIFY(vdev_metaslab_init(vd, txg) == 0);
3296                 vdev_config_dirty(vd);
3297         }
3298 }

3300 /*
3301  * Split a vdev.
3302  */
3303 void
3304 vdev_split(vdev_t *vd)
3305 {
3306         vdev_t *cvd, *pvd = vd->vdev_parent;

3308         vdev_remove_child(pvd, vd);
3309         vdev_compact_children(pvd);

3311         cvd = pvd->vdev_child[0];
3312         if (pvd->vdev_children == 1) {
3313                 vdev_remove_parent(cvd);
3314                 cvd->vdev_splitting = B_TRUE;
3315         }
3316         vdev_propagate_state(cvd);
3317 }

3319 void
3320 vdev_deadman(vdev_t *vd)
3321 {
3322         for (int c = 0; c < vd->vdev_children; c++) {
3323                 vdev_t *cvd = vd->vdev_child[c];

3325                 vdev_deadman(cvd);
3326         }

3328         if (vd->vdev_ops->vdev_op_leaf) {
3329                 vdev_queue_t *vq = &vd->vdev_queue;

3331                 mutex_enter(&vq->vq_lock);
```

```
3332                 if (avl_numnodes(&vq->vq_active_tree) > 0) {
3333                         spa_t *spa = vd->vdev_spa;
3334                         zio_t *fio;
3335                         uint64_t delta;
3337                         /*
3338                          * Look at the head of all the pending queues,
3339                          * if any I/O has been outstanding for longer than
3340                          * the spa_deadman_synctime we panic the system.
3341                          */
3342                         fio = avl_first(&vq->vq_active_tree);
3343                         delta = gethrtime() - fio->io_timestamp;
3344                         if (delta > spa_deadman_synctime(spa)) {
3345                                 zfs_dbgmsg("SLOW IO: zio timestamp %lluns, "
3346                                     "delta %lluns, last io %lluns",
3347                                     fio->io_timestamp, delta,
3348                                     vq->vq_io_complete_ts);
3349                                 fm_panic("I/O to pool '%s' appears to be "
3350                                     "hung.", spa_name(spa));
3351                         }
3352                 }
3353                 mutex_exit(&vq->vq_lock);
3354         }
3355 }
```

```
**********************************************************
   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_
```

```
629 int
630 zil_claim(dsl_pool_t *dp, dsl_dataset_t *ds, void *txarg)
630 zil_claim(const char *osname, void *txarg)
631 {
632         dmu_tx_t *tx = txarg;
633         uint64_t first_txg = dmu_tx_get_txg(tx);
634         zilog_t *zilog;
635         zil_header_t *zh;
636         objset_t *os;
637         int error;

639         error = dmu_objset_own_obj(dp, ds->ds_object,
640             DMU_OST_ANY, B_FALSE, FTAG, &os);
639         error = dmu_objset_own(osname, DMU_OST_ANY, B_FALSE, FTAG, &os);
641         if (error != 0) {
642                 /*
643                  * EBUSY indicates that the objset is inconsistent, in which
644                  * case it can not have a ZIL.
645                  */
646                 if (error != EBUSY) {
647                         cmn_err(CE_WARN, "can't open objset for %llu, error %u",
648                             (unsigned long long)ds->ds_object, error);
646                         cmn_err(CE_WARN, "can't open objset for %s, error %u",
647                             osname, error);
649                 }
650                 return (0);
651         }

653         zilog = dmu_objset_zil(os);
654         zh = zil_header_in_syncing_context(zilog);

656         if (spa_get_log_state(zilog->zl_spa) == SPA_LOG_CLEAR) {
657                 if (!BP_IS_HOLE(&zh->zh_log))
658                         zio_free_zil(zilog->zl_spa, first_txg, &zh->zh_log);
659                 BP_ZERO(&zh->zh_log);
660                 dsl_dataset_dirty(dmu_objset_ds(os), tx);
661                 dmu_objset_disown(os, FTAG);
662                 return (0);
663         }

665         /*
666          * Claim all log blocks if we haven't already done so, and remember
667          * the highest claimed sequence number.  This ensures that if we can
668          * read only part of the log now (e.g. due to a missing device),
669          * but we can read the entire log later, we will not try to replay
670          * or destroy beyond the last block we successfully claimed.
671          */
672         ASSERT3U(zh->zh_claim_txg, <=, first_txg);
673         if (zh->zh_claim_txg == 0 && !BP_IS_HOLE(&zh->zh_log)) {
```



```
674                 (void) zil_parse(zilog, zil_claim_log_block,
675                     zil_claim_log_record, tx, first_txg);
676                 zh->zh_claim_txg = first_txg;
677                 zh->zh_claim_blk_seq = zilog->zl_parse_blk_seq;
678                 zh->zh_claim_lr_seq = zilog->zl_parse_lr_seq;
679                 if (zilog->zl_parse_lr_count || zilog->zl_parse_blk_count > 1)
680                         zh->zh_flags |= ZIL_REPLAY_NEEDED;
681                 zh->zh_flags |= ZIL_CLAIM_LR_SEQ_VALID;
682                 dsl_dataset_dirty(dmu_objset_ds(os), tx);
683         }

685         ASSERT3U(first_txg, ==, (spa_last_synced_txg(zilog->zl_spa) + 1));
686         dmu_objset_disown(os, FTAG);
687         return (0);
688 }

690 /*
691  * Check the log by walking the log chain.
692  * Checksum errors are ok as they indicate the end of the chain.
693  * Any other error (no device or read failure) returns an error.
694  */
695 /* ARGSUSED */
696 #endif /* ! codereview */
697 int
698 zil_check_log_chain(dsl_pool_t *dp, dsl_dataset_t *ds, void *tx)
694 zil_check_log_chain(const char *osname, void *tx)
699 {
700         zilog_t *zilog;
701         objset_t *os;
702         blkptr_t *bp;
703         int error;

705         ASSERT(tx == NULL);

707         error = dmu_objset_from_ds(ds, &os);
703         error = dmu_objset_hold(osname, FTAG, &os);
708         if (error != 0) {
709                 cmn_err(CE_WARN, "can't open objset %llu, error %d",
710                     (unsigned long long)ds->ds_object, error);
705                 cmn_err(CE_WARN, "can't open objset for %s", osname);
711                 return (0);
712         }

714         zilog = dmu_objset_zil(os);
715         bp = (blkptr_t *)&zilog->zl_header->zh_log;

717         /*
718          * Check the first block and determine if it's on a log device
719          * which may have been removed or faulted prior to loading this
720          * pool.  If so, there's no point in checking the rest of the log
721          * as its content should have already been synced to the pool.
722          */
723         if (!BP_IS_HOLE(bp)) {
724                 vdev_t *vd;
725                 boolean_t valid = B_TRUE;

727                 spa_config_enter(os->os_spa, SCL_STATE, FTAG, RW_READER);
728                 vd = vdev_lookup_top(os->os_spa, DVA_GET_VDEV(&bp->blk_dva[0]));
729                 if (vd->vdev_islog && vdev_is_dead(vd))
730                         valid = vdev_log_state_valid(vd);
731                 spa_config_exit(os->os_spa, SCL_STATE, FTAG);

733                 if (!valid)
728                 if (!valid) {
729                         dmu_objset_rele(os, FTAG);
734                         return (0);
```

```
 735         }
 732         }

 737         /*
 738          * Because tx == NULL, zil_claim_log_block() will not actually claim
 739          * any blocks, but just determine whether it is possible to do so.
 740          * In addition to checking the log chain, zil_claim_log_block()
 741          * will invoke zio_claim() with a done func of spa_claim_notify(),
 742          * which will update spa_max_claim_txg.  See spa_load() for details.
 743          */
 744         error = zil_parse(zilog, zil_claim_log_block, zil_claim_log_record, tx,
 745             zilog->zl_header->zh_claim_txg ? -1ULL : spa_first_txg(os->os_spa));

 744         dmu_objset_rele(os, FTAG);

 747         return ((error == ECKSUM || error == ENOENT) ? 0 : error);
 748 }
```

*unchanged_portion_omitted_*