

new/usr/src/cmd/zdb/Makefile.com

1

```
*****
1754 Thu Jul 26 14:18:53 2012
new/usr/src/cmd/zdb/Makefile.com
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 #
23 # Copyright 2009 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #
26 # Copyright (c) 2012 by Delphix. All rights reserved.
27 #
28 #
29 PROG:sh=      cd ../; basename `pwd`
30 SRCS= ../$(PROG).c ../zdb_il.c
31 OBJS= $(PROG).o zdb_il.o
32 #
33 include ../../Makefile.cmd
34 include ../../Makefile.ctf
35 #
36 INCS += -I../../lib/libzpool/common
37 INCS += -I../../uts/common/fs/zfs
38 INCS += -I../../common/zfs
39 #
40 LDLIBS += -lzpool -lumem -lnvpair -lzfs -lavl
41 #
42 C99MODE=      -xc99=%all
43 C99LMODE=     -Xc99=%all
44 #
45 CFLAGS += $(CCVERBOSE)
46 CFLAGS64 += $(CCVERBOSE)
47 CPPFLAGS += -D_LARGEFILE64_SOURCE=1 -D_REENTRANT $(INCS) -DDEBUG
48 CPPFLAGS += -D_LARGEFILE64_SOURCE=1 -D_REENTRANT $(INCS)
49 # lint complains about unused _umem_* functions
50 LINTFLAGS += -xerroff=E_NAME_DEF_NOT_USED2
51 LINTFLAGS64 += -xerroff=E_NAME_DEF_NOT_USED2
52 #
53 .KEEP_STATE:
54 #
55 all: $(PROG)
56 #
57 $(PROG): $(OBJS)
58     $(LINK.c) -o $(PROG) $(OBJS) $(LDLIBS)
59     $(POST_PROCESS)
```

new/usr/src/cmd/zdb/Makefile.com

2

```
61 clean:
62 #
63 lint: lint_SRCS
64 #
65 include ../../Makefile.targ
66 #
67 %.o: ../%.c
68     $(COMPILE.c) $<
69     $(POST_PROCESS_0)
```

```

*****
83672 Thu Jul 26 14:18:54 2012
new/usr/src/cmd/zdb/zdb.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****

```

\_\_\_\_\_unchanged\_portion\_omitted\_\_\_\_\_

```

515 static void
516 dump_spacemap(objset_t *os, space_map_obj_t *smo, space_map_t *sm)
517 {
518     uint64_t alloc, offset, entry;
519     uint8_t mapshift = sm->sm_shift;
520     uint64_t mapstart = sm->sm_start;
521     char *ddata[] = { "ALLOC", "FREE", "CONDENSE", "INVALID",
522                     "INVALID", "INVALID", "INVALID", "INVALID" };
523
524     if (smo->smo_object == 0)
525         return;
526
527     /*
528      * Print out the freelist entries in both encoded and decoded form.
529      */
530     alloc = 0;
531     for (offset = 0; offset < smo->smo_objsize; offset += sizeof (entry)) {
532         VERIFY0(dmu_read(os, smo->smo_object, offset,
533             VERIFY3U(0, ==, dmu_read(os, smo->smo_object, offset,
534                 sizeof (entry), &entry, DMU_READ_PREFETCH));
535         if (SM_DEBUG_DECODE(entry)) {
536             (void) printf("\t [%6llu] %s: txg %llu, pass %llu\n",
537                 (u_longlong_t)(offset / sizeof (entry)),
538                 ddata[SM_DEBUG_ACTION_DECODE(entry)],
539                 (u_longlong_t)SM_DEBUG_TXG_DECODE(entry),
540                 (u_longlong_t)SM_DEBUG_SYNCPASS_DECODE(entry));
541         } else {
542             (void) printf("\t [%6llu] %c range:"
543                 "\t %010llx-%010llx size: %06llx\n",
544                 (u_longlong_t)(offset / sizeof (entry)),
545                 SM_TYPE_DECODE(entry) == SM_ALLOC ? 'A' : 'F',
546                 (u_longlong_t)((SM_OFFSET_DECODE(entry) <<
547                     mapshift) + mapstart),
548                 (u_longlong_t)((SM_OFFSET_DECODE(entry) <<
549                     mapshift) + mapstart + (SM_RUN_DECODE(entry) <<
550                     mapshift)),
551                 (u_longlong_t)(SM_RUN_DECODE(entry) << mapshift));
552             if (SM_TYPE_DECODE(entry) == SM_ALLOC)
553                 alloc += SM_RUN_DECODE(entry) << mapshift;
554             else
555                 alloc -= SM_RUN_DECODE(entry) << mapshift;
556         }
557     }
558     if (alloc != smo->smo_alloc) {
559         (void) printf("space_map_object alloc (%llu) INCONSISTENT "
560             "with space map summary (%llu)\n",
561             (u_longlong_t)smo->smo_alloc, (u_longlong_t)alloc);
562     }

```

\_\_\_\_\_unchanged\_portion\_omitted\_\_\_\_\_

```

1181 static void
1182 dump_bptree(objset_t *os, uint64_t obj, char *name)
1183 {
1184     char bytes[32];
1185     bptree_phys_t *bt;
1186     dmu_buf_t *db;
1187
1188     if (dump_opt['d'] < 3)

```

```

1189         return;
1190
1191     VERIFY0(dmu_bonus_hold(os, obj, FTAG, &db));
1192     VERIFY3U(0, ==, dmu_bonus_hold(os, obj, FTAG, &db));
1193     bt = db->db_data;
1194     zdb_nicenum(bt->bt_bytes, bytes);
1195     (void) printf("\n %s: %llu datasets, %s\n",
1196         name, (unsigned long long)(bt->bt_end - bt->bt_begin), bytes);
1197     dmu_buf_rele(db, FTAG);
1198
1199     if (dump_opt['d'] < 5)
1200         return;
1201
1202     (void) printf("\n");
1203
1204     (void) bptree_iterate(os, obj, B_FALSE, dump_bptree_cb, NULL, NULL);
1205 }
1206 _____unchanged_portion_omitted_____
1207
1221 static int
1222 dump_block_stats(spa_t *spa)
1223 {
1224     zdb_cb_t zcb = { 0 };
1225     zdb_blkstats_t *zb, *tzb;
1226     uint64_t norm_alloc, norm_space, total_alloc, total_found;
1227     int flags = TRAVERSE_PRE | TRAVERSE_PREFETCH_METADATA | TRAVERSE_HARD;
1228     int leaks = 0;
1229
1230     (void) printf("\nTraversing all blocks %s%s%s%s%s...\n",
1231         (dump_opt['c'] || !dump_opt['L']) ? "to verify " : "",
1232         (dump_opt['c'] == 1) ? "metadata " : "",
1233         dump_opt['c'] ? "checksums " : "",
1234         (dump_opt['c'] && !dump_opt['L']) ? "and verify " : "",
1235         !dump_opt['L'] ? "nothing leaked " : "");
1236
1237     /*
1238      * Load all space maps as SM_ALLOC maps, then traverse the pool
1239      * claiming each block we discover. If the pool is perfectly
1240      * consistent, the space maps will be empty when we're done.
1241      * Anything left over is a leak; any block we can't claim (because
1242      * it's not part of any space map) is a double allocation,
1243      * reference to a freed block, or an unclaimed log block.
1244      */
1245     zdb_leak_init(spa, &zcb);
1246
1247     /*
1248      * If there's a deferred-free bplist, process that first.
1249      */
1250     (void) bpobj_iterate_nofree(&spa->spa_deferred_bpobj,
1251         count_block_cb, &zcb, NULL);
1252     (void) bpobj_iterate_nofree(&spa->spa_dsl_pool->dp_free_bpobj,
1253         count_block_cb, &zcb, NULL);
1254     if (spa_feature_is_active(spa,
1255         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
1256         VERIFY0(bptree_iterate(spa->spa_meta_objset,
1257             VERIFY3U(0, ==, bptree_iterate(spa->spa_meta_objset,
1258                 spa->spa_dsl_pool->dp_bptree_obj, B_FALSE, count_block_cb,
1259                 &zcb, NULL));
1260     }
1261
1262     if (dump_opt['c'] > 1)
1263         flags |= TRAVERSE_PREFETCH_DATA;
1264
1265     zcb.zcb_haderrors |= traverse_pool(spa, 0, flags, zdb_blkptr_cb, &zcb);
1266
1267     if (zcb.zcb_haderrors) {

```

```

2287     (void) printf("\nError counts:\n\n");
2288     (void) printf("\t\t%5s %s\n", "errno", "count");
2289     for (int e = 0; e < 256; e++) {
2290         if (zcb.zcb_errors[e] != 0) {
2291             (void) printf("\t\t%5d %llu\n",
2292                 e, (u_longlong_t)zcb.zcb_errors[e]);
2293         }
2294     }
2295 }
2297 /*
2298  * Report any leaked segments.
2299  */
2300 zdb_leak_fini(spa);
2302
2303 tzb = &zcb.zcb_type[ZB_TOTAL][ZDB_OT_TOTAL];
2304
2305 norm_alloc = metaslab_class_get_alloc(spa_normal_class(spa));
2306 norm_space = metaslab_class_get_space(spa_normal_class(spa));
2307
2308 total_alloc = norm_alloc + metaslab_class_get_alloc(spa_log_class(spa));
2309 total_found = tzb->zb_asize - zcb.zcb_dedup_asize;
2310
2311 if (total_found == total_alloc) {
2312     if (!dump_opt['L'])
2313         (void) printf("\n\tNo leaks (block sum matches space
2314             " maps exactly)\n");
2315 } else {
2316     (void) printf("block traversal size %llu != alloc %llu "
2317         "(%s %lld)\n",
2318         (u_longlong_t)total_found,
2319         (u_longlong_t)total_alloc,
2320         (dump_opt['L'] ? "unreachable" : "leaked",
2321         (longlong_t)(total_alloc - total_found));
2322     leaks = 1;
2323 }
2324
2325 if (tzb->zb_count == 0)
2326     return (2);
2327
2328 (void) printf("\n");
2329 (void) printf("\t\tbp count: %10llu\n",
2330     (u_longlong_t)tzb->zb_count);
2331 (void) printf("\t\tbp logical: %10llu avg: %6llu\n",
2332     (u_longlong_t)tzb->zb_lsize,
2333     (u_longlong_t)(tzb->zb_lsize / tzb->zb_count));
2334 (void) printf("\t\tbp physical: %10llu avg:"
2335     " %6llu compression: %6.2f\n",
2336     (u_longlong_t)tzb->zb_psize,
2337     (u_longlong_t)(tzb->zb_psize / tzb->zb_count),
2338     (double)tzb->zb_lsize / tzb->zb_psize);
2339 (void) printf("\t\tbp allocated: %10llu avg:"
2340     " %6llu compression: %6.2f\n",
2341     (u_longlong_t)tzb->zb_asize,
2342     (u_longlong_t)(tzb->zb_asize / tzb->zb_count),
2343     (double)tzb->zb_lsize / tzb->zb_asize);
2344 (void) printf("\t\tbp deduped: %10llu ref>1:"
2345     " %6llu deduplication: %6.2f\n",
2346     (u_longlong_t)zcb.zcb_dedup_asize,
2347     (u_longlong_t)zcb.zcb_dedup_blocks,
2348     (double)zcb.zcb_dedup_asize / tzb->zb_asize + 1.0);
2349 (void) printf("\t\tSPA allocated: %10llu used: %5.2f%%\n",
2350     (u_longlong_t)norm_alloc, 100.0 * norm_alloc / norm_space);
2351
2352 if (dump_opt['b'] >= 2) {
2353     int l, t, level;

```

```

2353     (void) printf("\nBlocks\tLSIZE\tPSIZE\tASIZE"
2354         "\t\t avg\t comp\t%%Total\tType\n");
2355
2356     for (t = 0; t <= ZDB_OT_TOTAL; t++) {
2357         char csize[32], lsize[32], psize[32], asize[32];
2358         char avg[32];
2359         char *typename;
2360
2361         if (t < DMU_OT_NUMTYPES)
2362             typename = dmu_ot[t].ot_name;
2363         else
2364             typename = zdb_ot_extname[t - DMU_OT_NUMTYPES];
2365
2366         if (zcb.zcb_type[ZB_TOTAL][t].zb_asize == 0) {
2367             (void) printf("%6s\t%5s\t%5s\t%5s"
2368                 "\t\t%5s\t%5s\t%6s\t%5s\n",
2369                 "-",
2370                 "-",
2371                 "-",
2372                 "-",
2373                 "-",
2374                 "-",
2375                 "-",
2376                 typename);
2377             continue;
2378         }
2379
2380         for (l = ZB_TOTAL - 1; l >= -1; l--) {
2381             level = (l == -1 ? ZB_TOTAL : l);
2382             zb = &zcb.zcb_type[level][t];
2383
2384             if (zb->zb_asize == 0)
2385                 continue;
2386
2387             if (dump_opt['b'] < 3 && level != ZB_TOTAL)
2388                 continue;
2389
2390             if (level == 0 && zb->zb_asize ==
2391                 zcb.zcb_type[ZB_TOTAL][t].zb_asize)
2392                 continue;
2393
2394             zdb_nicenum(zb->zb_count, csize);
2395             zdb_nicenum(zb->zb_lsize, lsize);
2396             zdb_nicenum(zb->zb_psize, psize);
2397             zdb_nicenum(zb->zb_asize, asize);
2398             zdb_nicenum(zb->zb_asize / zb->zb_count, avg);
2399
2400             (void) printf("%6s\t%5s\t%5s\t%5s\t%5s"
2401                 "\t\t%5.2f\t%6.2f\t",
2402                 csize, lsize, psize, asize, avg,
2403                 (double)zb->zb_lsize / zb->zb_psize,
2404                 100.0 * zb->zb_asize / tzb->zb_asize);
2405
2406             if (level == ZB_TOTAL)
2407                 (void) printf("%s\n", typename);
2408             else
2409                 (void) printf(" L%d %s\n",
2410                     level, typename);
2411         }
2412     }
2413 }
2414
2415 (void) printf("\n");
2416
2417 if (leaks)
2418     return (2);

```

new/usr/src/cmd/zdb/zdb.c

5

```
2420         if (zcb.zcb_haderrors)
2421             return (3);
2423         return (0);
2424     }
unchanged_portion_omitted
```

new/usr/src/cmd/zhack/Makefile.com

1

```
*****
1573 Thu Jul 26 14:18:55 2012
new/usr/src/cmd/zhack/Makefile.com
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 #
23 # Copyright 2009 Sun Microsystems, Inc. All rights reserved.
24 # Use is subject to license terms.
25 #
26 #
27 #
28 # Copyright (c) 2012 by Delphix. All rights reserved.
29 #
30 #
31 PROG= zhack
32 SRCS= ../$(PROG).c
33 OBJS= $(PROG).o
34 #
35 include ../../Makefile.cmd
36 include ../../Makefile.ctf
37 #
38 INCS += -I../../lib/libzpool/common
39 INCS += -I../../uts/common/fs/zfs
40 INCS += -I../../common/zfs
41 #
42 LDLIBS += -lzpool -lumem -lnvpair -lzfs
43 #
44 C99MODE= -xc99=%all
45 C99LMODE= -Xc99=%all
46 #
47 CFLAGS += $(CCVERBOSE)
48 CFLAGS64 += $(CCVERBOSE)
49 CPPFLAGS += -D_LARGEFILE64_SOURCE=1 -D_REENTRANT -DDEBUG $(INCS)
50 CPPFLAGS += -D_LARGEFILE64_SOURCE=1 -D_REENTRANT $(INCS)
51 .KEEP_STATE:
52 #
53 all: $(PROG)
54 #
55 $(PROG): $(OBJS)
56 $(LINK.c) -o $(PROG) $(OBJS) $(LDLIBS)
57 $(POST_PROCESS)
58 #
59 clean:
```

new/usr/src/cmd/zhack/Makefile.com

2

```
61 lint: lint_SRCS
62 #
63 include ../../Makefile.targ
64 #
65 %.o: ../%.c
66 $(COMPILE.c) $<
67 $(POST_PROCESS_0)
```

```

*****
12718 Thu Jul 26 14:18:56 2012
new/usr/src/cmd/zhack/zhack.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

287 static void
288 zhack_do_feature_enable(int argc, char **argv)
289 {
290     char c;
291     char *desc, *target;
292     spa_t *spa;
293     objset_t *mos;
294     zfeature_info_t feature;
295     zfeature_info_t *nodeps[] = { NULL };

297     /*
298      * Features are not added to the pool's label until their refcounts
299      * are incremented, so fi_mos can just be left as false for now.
300      */
301     desc = NULL;
302     feature.fi_uname = "zhack";
303     feature.fi_mos = B_FALSE;
304     feature.fi_can_readonly = B_FALSE;
305     feature.fi_depends = nodeps;

307     optind = 1;
308     while ((c = getopt(argc, argv, "rmd:")) != -1) {
309         switch (c) {
310             case 'r':
311                 feature.fi_can_readonly = B_TRUE;
312                 break;
313             case 'd':
314                 desc = strdup(optarg);
315                 break;
316             default:
317                 usage();
318                 break;
319         }
320     }

322     if (desc == NULL)
323         desc = strdup("zhack injected");
324     feature.fi_desc = desc;

326     argc -= optind;
327     argv += optind;

329     if (argc < 2) {
330         (void) fprintf(stderr, "error: missing feature or pool name\n");
331         usage();
332     }
333     target = argv[0];
334     feature.fi_guid = argv[1];

336     if (!zfeature_is_valid_guid(feature.fi_guid))
337         fatal("invalid feature guid: %s", feature.fi_guid);

339     zhack_spa_open(target, B_FALSE, FTAG, &spa);
340     mos = spa->spa_meta_objset;

342     if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
343         fatal("'s' is a real feature, will not enable");
344     if (0 == zap_contains(mos, spa->spa_feat_desc_obj, feature.fi_guid))
345         fatal("feature already enabled: %s", feature.fi_guid);

```

```

347     VERIFY0(dsl_sync_task_do(spa->spa_dsl_pool, NULL,
347     VERIFY3U(0, ==, dsl_sync_task_do(spa->spa_dsl_pool, NULL,
348     feature_enable_sync, spa, &feature, 5)));

350     spa_close(spa, FTAG);

352     free(desc);
353 }
_____unchanged_portion_omitted_____

377 static void
378 zhack_do_feature_ref(int argc, char **argv)
379 {
380     char c;
381     char *target;
382     boolean_t decr = B_FALSE;
383     spa_t *spa;
384     objset_t *mos;
385     zfeature_info_t feature;
386     zfeature_info_t *nodeps[] = { NULL };

388     /*
389      * fi_desc does not matter here because it was written to disk
390      * when the feature was enabled, but we need to properly set the
391      * feature for read or write based on the information we read off
392      * disk later.
393      */
394     feature.fi_uname = "zhack";
395     feature.fi_mos = B_FALSE;
396     feature.fi_desc = NULL;
397     feature.fi_depends = nodeps;

399     optind = 1;
400     while ((c = getopt(argc, argv, "md")) != -1) {
401         switch (c) {
402             case 'm':
403                 feature.fi_mos = B_TRUE;
404                 break;
405             case 'd':
406                 decr = B_TRUE;
407                 break;
408             default:
409                 usage();
410                 break;
411         }
412     }
413     argc -= optind;
414     argv += optind;

416     if (argc < 2) {
417         (void) fprintf(stderr, "error: missing feature or pool name\n");
418         usage();
419     }
420     target = argv[0];
421     feature.fi_guid = argv[1];

423     if (!zfeature_is_valid_guid(feature.fi_guid))
424         fatal("invalid feature guid: %s", feature.fi_guid);

426     zhack_spa_open(target, B_FALSE, FTAG, &spa);
427     mos = spa->spa_meta_objset;

429     if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
430         fatal("'s' is a real feature, will not change refcount");

```

```
432     if (0 == zap_contains(mos, spa->spa_feat_for_read_obj,
433         feature.fi_guid)) {
434         feature.fi_can_readonly = B_FALSE;
435     } else if (0 == zap_contains(mos, spa->spa_feat_for_write_obj,
436         feature.fi_guid)) {
437         feature.fi_can_readonly = B_TRUE;
438     } else {
439         fatal("feature is not enabled: %s", feature.fi_guid);
440     }
441
442     if (decr && !spa_feature_is_active(spa, &feature))
443         fatal("feature refcount already 0: %s", feature.fi_guid);
444
445     VERIFY0(dsl_sync_task_do(spa->spa_dsl_pool, NULL,
446         VERIFY3U(0, ==, dsl_sync_task_do(spa->spa_dsl_pool, NULL,
447         decr ? feature_decr_sync : feature_incr_sync, spa, &feature, 5)));
448     spa_close(spa, FTAG);
449 }
unchanged_portion_omitted
```

new/usr/src/cmd/ztest/Makefile.com

1

```
*****
1706 Thu Jul 26 14:18:57 2012
new/usr/src/cmd/ztest/Makefile.com
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 PROG= ztest
27 OBJS= $(PROG).o
28 SRCS= $(OBJS:%.o=../%.c)

30 include ../../Makefile.cmd
31 include ../../Makefile.ctf

33 INCS += -I../../lib/libzpool/common
34 INCS += -I../../uts/common/fs/zfs
35 INCS += -I../../common/zfs

37 LDLIBS += -lumem -lzpool -lcmdutils -lm -lnvpair

39 C99MODE= -xc99=%all
40 C99LMODE= -Xc99=%all
41 CFLAGS += -g $(CCVERBOSE)
42 CFLAGS64 += -g $(CCVERBOSE)
43 CPPFLAGS += -D_LARGEFILE64_SOURCE=1 -D_REENTRANT $(INCS) -DDEBUG
43 CPPFLAGS += -D_LARGEFILE64_SOURCE=1 -D_REENTRANT $(INCS)

45 # lint complains about unused _umem_* functions
46 LINTFLAGS += -xeroff=E_NAME_DEF_NOT_USED2
47 LINTFLAGS64 += -xeroff=E_NAME_DEF_NOT_USED2

49 .KEEP_STATE:

51 all: $(PROG)

53 $(PROG): $(OBJS)
54     $(LINK.c) -o $(PROG) $(OBJS) $(LDLIBS)
55     $(POST_PROCESS)

57 clean:

59 lint:    lint_SRCS
```

new/usr/src/cmd/ztest/Makefile.com

2

```
61 include ../../Makefile.targ

63 %.o: ../%.c
64     $(COMPILE.c) $<
65     $(POST_PROCESS_0)
```



```

*****
152861 Thu Jul 26 14:18:58 2012
new/usr/src/cmd/ztest/ztest.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

957 static int
958 ztest_dsl_prop_set_uint64(char *osname, zfs_prop_t prop, uint64_t value,
959     boolean_t inherit)
960 {
961     const char *propname = zfs_prop_to_name(prop);
962     const char *valname;
963     char setpoint[MAXPATHLEN];
964     uint64_t curval;
965     int error;

967     error = dsl_prop_set(osname, propname,
968         (inherit ? ZPROP_SRC_NONE : ZPROP_SRC_LOCAL),
969         sizeof (value), 1, &value);

971     if (error == ENOSPC) {
972         ztest_record_enospc(FTAG);
973         return (error);
974     }
975     ASSERT0(error);
975     ASSERT3U(error, ==, 0);

977     VERIFY3U(dsl_prop_get(osname, propname, sizeof (curval),
978         1, &curval, setpoint), ==, 0);

980     if (ztest_opts.zo_verbose >= 6) {
981         VERIFY(zfs_prop_index_to_string(prop, curval, &valname) == 0);
982         (void) printf("%s %s = %s at '%s'\n",
983             osname, propname, valname, setpoint);
984     }

986     return (error);
987 }

989 static int
990 ztest_spa_prop_set_uint64(zpool_prop_t prop, uint64_t value)
991 {
992     spa_t *spa = ztest_spa;
993     nvlist_t *props = NULL;
994     int error;

996     VERIFY(nvlist_alloc(&props, NV_UNIQUE_NAME, 0) == 0);
997     VERIFY(nvlist_add_uint64(props, zpool_prop_to_name(prop), value) == 0);

999     error = spa_prop_set(spa, props);

1001     nvlist_free(props);

1003     if (error == ENOSPC) {
1004         ztest_record_enospc(FTAG);
1005         return (error);
1006     }
1007     ASSERT0(error);
1007     ASSERT3U(error, ==, 0);

1009     return (error);
1010 }
_____unchanged_portion_omitted_____

1347 /*

```

```

1348 * ZIL replay ops
1349 */
1350 static int
1351 ztest_replay_create(ztest_ds_t *zd, lr_create_t *lr, boolean_t byteswap)
1352 {
1353     char *name = (void *) (lr + 1); /* name follows lr */
1354     objset_t *os = zd->zds_os;
1355     ztest_block_tag_t *bbt;
1356     dmu_buf_t *db;
1357     dmu_tx_t *tx;
1358     uint64_t txg;
1359     int error = 0;

1361     if (byteswap)
1362         byteswap_uint64_array(lr, sizeof (*lr));

1364     ASSERT(lr->lr_doid == ZTEST_DIROBJ);
1365     ASSERT(name[0] != '\0');

1367     tx = dmu_tx_create(os);

1369     dmu_tx_hold_zap(tx, lr->lr_doid, B_TRUE, name);

1371     if (lr->lrz_type == DMU_OT_ZAP_OTHER) {
1372         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, B_TRUE, NULL);
1373     } else {
1374         dmu_tx_hold_bonus(tx, DMU_NEW_OBJECT);
1375     }

1377     txg = ztest_tx_assign(tx, TXG_WAIT, FTAG);
1378     if (txg == 0)
1379         return (ENOSPC);

1381     ASSERT(dmu_objset_zil(os)->zil_replay == !!lr->lr_foid);

1383     if (lr->lrz_type == DMU_OT_ZAP_OTHER) {
1384         if (lr->lr_foid == 0) {
1385             lr->lr_foid = zap_create(os,
1386                 lr->lrz_type, lr->lrz_bonustype,
1387                 lr->lrz_bonuslen, tx);
1388         } else {
1389             error = zap_create_claim(os, lr->lr_foid,
1390                 lr->lrz_type, lr->lrz_bonustype,
1391                 lr->lrz_bonuslen, tx);
1392         }
1393     } else {
1394         if (lr->lr_foid == 0) {
1395             lr->lr_foid = dmu_object_alloc(os,
1396                 lr->lrz_type, 0, lr->lrz_bonustype,
1397                 lr->lrz_bonuslen, tx);
1398         } else {
1399             error = dmu_object_claim(os, lr->lr_foid,
1400                 lr->lrz_type, 0, lr->lrz_bonustype,
1401                 lr->lrz_bonuslen, tx);
1402         }
1403     }

1405     if (error) {
1406         ASSERT3U(error, ==, EEXIST);
1407         ASSERT(zd->zds_zilog->zil_replay);
1408         dmu_tx_commit(tx);
1409         return (error);
1410     }

1412     ASSERT(lr->lr_foid != 0);

```

```

1414     if (lr->lrz_type != DMU_OT_ZAP_OTHER)
1415         VERIFY0(dmu_object_set_blocksize(os, lr->lr_foid,
1416         VERIFY3U(0, ==, dmu_object_set_blocksize(os, lr->lr_foid,
1417         lr->lrz_blocksize, lr->lrz_ibshift, tx));
1418     VERIFY0(dmu_bonus_hold(os, lr->lr_foid, FTAG, &db));
1418     VERIFY3U(0, ==, dmu_bonus_hold(os, lr->lr_foid, FTAG, &db));
1419     bbt = ztest_bt_bonus(db);
1420     dmu_buf_will_dirty(db, tx);
1421     ztest_bt_generate(bbt, os, lr->lr_foid, -1ULL, lr->lr_gen, txg, txg);
1422     dmu_buf_rele(db, FTAG);
1424     VERIFY0(zap_add(os, lr->lr_doid, name, sizeof (uint64_t), 1,
1424     VERIFY3U(0, ==, zap_add(os, lr->lr_doid, name, sizeof (uint64_t), 1,
1425     &lr->lr_foid, tx));
1427     (void) ztest_log_create(zd, tx, lr);
1429     dmu_tx_commit(tx);
1431     return (0);
1432 }
1434 static int
1435 ztest_replay_remove(ztest_ds_t *zd, lr_remove_t *lr, boolean_t byteswap)
1436 {
1437     char *name = (void *) (lr + 1);          /* name follows lr */
1438     objset_t *os = zd->zd_os;
1439     dmu_object_info_t doi;
1440     dmu_tx_t *tx;
1441     uint64_t object, txg;
1443     if (byteswap)
1444         byteswap_uint64_array(lr, sizeof (*lr));
1446     ASSERT(lr->lr_doid == ZTEST_DIROBJ);
1447     ASSERT(name[0] != '\0');
1449     VERIFY0(
1449     VERIFY3U(0, ==,
1450     zap_lookup(os, lr->lr_doid, name, sizeof (object), 1, &object));
1451     ASSERT(object != 0);
1453     ztest_object_lock(zd, object, RL_WRITER);
1455     VERIFY0(dmu_object_info(os, object, &doi));
1455     VERIFY3U(0, ==, dmu_object_info(os, object, &doi));
1457     tx = dmu_tx_create(os);
1459     dmu_tx_hold_zap(tx, lr->lr_doid, B_FALSE, name);
1460     dmu_tx_hold_free(tx, object, 0, DMU_OBJECT_END);
1462     txg = ztest_tx_assign(tx, TXG_WAIT, FTAG);
1463     if (txg == 0) {
1464         ztest_object_unlock(zd, object);
1465         return (ENOSPC);
1466     }
1468     if (doi.doi_type == DMU_OT_ZAP_OTHER) {
1469         VERIFY0(zap_destroy(os, object, tx));
1469         VERIFY3U(0, ==, zap_destroy(os, object, tx));
1470     } else {
1471         VERIFY0(dmu_object_free(os, object, tx));
1471         VERIFY3U(0, ==, dmu_object_free(os, object, tx));
1472     }

```

```

1474     VERIFY0(zap_remove(os, lr->lr_doid, name, tx));
1474     VERIFY3U(0, ==, zap_remove(os, lr->lr_doid, name, tx));
1476     (void) ztest_log_remove(zd, tx, lr, object);
1478     dmu_tx_commit(tx);
1480     ztest_object_unlock(zd, object);
1482     return (0);
1483 }
1485 static int
1486 ztest_replay_write(ztest_ds_t *zd, lr_write_t *lr, boolean_t byteswap)
1487 {
1488     objset_t *os = zd->zd_os;
1489     void *data = lr + 1;                    /* data follows lr */
1490     uint64_t offset, length;
1491     ztest_block_tag_t *bt = data;
1492     ztest_block_tag_t *bbt;
1493     uint64_t gen, txg, lrtxg, crtxg;
1494     dmu_object_info_t doi;
1495     dmu_tx_t *tx;
1496     dmu_buf_t *db;
1497     arc_buf_t *abuf = NULL;
1498     rl_t *rl;
1500     if (byteswap)
1501         byteswap_uint64_array(lr, sizeof (*lr));
1503     offset = lr->lr_offset;
1504     length = lr->lr_length;
1506     /* If it's a dmu_sync() block, write the whole block */
1507     if (lr->lr_common.lrc_reclen == sizeof (lr_write_t)) {
1508         uint64_t blocksize = BP_GET_LSIZE(&lr->lr_blkptr);
1509         if (length < blocksize) {
1510             offset -= offset % blocksize;
1511             length = blocksize;
1512         }
1513     }
1515     if (bt->bt_magic == BSWAP_64(BT_MAGIC))
1516         byteswap_uint64_array(bt, sizeof (*bt));
1518     if (bt->bt_magic != BT_MAGIC)
1519         bt = NULL;
1521     ztest_object_lock(zd, lr->lr_foid, RL_READER);
1522     rl = ztest_range_lock(zd, lr->lr_foid, offset, length, RL_WRITER);
1524     VERIFY0(dmu_bonus_hold(os, lr->lr_foid, FTAG, &db));
1524     VERIFY3U(0, ==, dmu_bonus_hold(os, lr->lr_foid, FTAG, &db));
1526     dmu_object_info_from_db(db, &doi);
1528     bbt = ztest_bt_bonus(db);
1529     ASSERT3U(bbt->bt_magic, ==, BT_MAGIC);
1530     gen = bbt->bt_gen;
1531     crtxg = bbt->bt_crtxg;
1532     lrtxg = lr->lr_common.lrc_txg;
1534     tx = dmu_tx_create(os);
1536     dmu_tx_hold_write(tx, lr->lr_foid, offset, length);

```

```

1538     if (ztest_random(8) == 0 && length == doi.doi_data_block_size &&
1539         P2PHASE(offset, length) == 0)
1540         abuf = dmu_request_arcbuf(db, length);

1542     txg = ztest_tx_assign(tx, TXG_WAIT, FTAG);
1543     if (txg == 0) {
1544         if (abuf != NULL)
1545             dmu_return_arcbuf(abuf);
1546         dmu_buf_rele(db, FTAG);
1547         ztest_range_unlock(rl);
1548         ztest_object_unlock(zd, lr->lr_foid);
1549         return (ENOSPC);
1550     }

1552     if (bt != NULL) {
1553         /*
1554          * Usually, verify the old data before writing new data --
1555          * but not always, because we also want to verify correct
1556          * behavior when the data was not recently read into cache.
1557          */
1558         ASSERT(offset % doi.doi_data_block_size == 0);
1559         if (ztest_random(4) != 0) {
1560             int prefetch = ztest_random(2) ?
1561                 DMU_READ_PREFETCH : DMU_READ_NO_PREFETCH;
1562             ztest_block_tag_t rbt;

1564             VERIFY(dmu_read(os, lr->lr_foid, offset,
1565                 sizeof (rbt), &rbt, prefetch) == 0);
1566             if (rbt.bt_magic == BT_MAGIC) {
1567                 ztest_bt_verify(&rbt, os, lr->lr_foid,
1568                     offset, gen, txg, crtngx);
1569             }
1570         }

1572         /*
1573          * Writes can appear to be newer than the bonus buffer because
1574          * the ztest_get_data() callback does a dmu_read() of the
1575          * open-context data, which may be different than the data
1576          * as it was when the write was generated.
1577          */
1578         if (zd->zilog->zl_replay) {
1579             ztest_bt_verify(bt, os, lr->lr_foid, offset,
1580                 MAX(gen, bt->bt_gen), MAX(txg, lrtxg),
1581                 bt->bt_crtxg);
1582         }

1584         /*
1585          * Set the bt's gen/txg to the bonus buffer's gen/txg
1586          * so that all of the usual ASSERTs will work.
1587          */
1588         ztest_bt_generate(bt, os, lr->lr_foid, offset, gen, txg, crtngx);
1589     }

1591     if (abuf == NULL) {
1592         dmu_write(os, lr->lr_foid, offset, length, data, tx);
1593     } else {
1594         bcopy(data, abuf->b_data, length);
1595         dmu_assign_arcbuf(db, offset, abuf, tx);
1596     }

1598     (void) ztest_log_write(zd, tx, lr);

1600     dmu_buf_rele(db, FTAG);

1602     dmu_tx_commit(tx);

```

```

1604     ztest_range_unlock(rl);
1605     ztest_object_unlock(zd, lr->lr_foid);

1607     return (0);
1608 }
    unchanged_portion_omitted

1649 static int
1650 ztest_replay_setattr(ztest_ds_t *zd, lr_setattr_t *lr, boolean_t byteswap)
1651 {
1652     objset_t *os = zd->zd_os;
1653     dmu_tx_t *tx;
1654     dmu_buf_t *db;
1655     ztest_block_tag_t *bbt;
1656     uint64_t txg, lrtxg, crtngx;

1658     if (byteswap)
1659         byteswap_uint64_array(lr, sizeof (*lr));

1661     ztest_object_lock(zd, lr->lr_foid, RL_WRITER);

1663     VERIFY0(dmu_bonus_hold(os, lr->lr_foid, FTAG, &db));
1664     VERIFY3U(0, ==, dmu_bonus_hold(os, lr->lr_foid, FTAG, &db));

1665     tx = dmu_tx_create(os);
1666     dmu_tx_hold_bonus(tx, lr->lr_foid);

1668     txg = ztest_tx_assign(tx, TXG_WAIT, FTAG);
1669     if (txg == 0) {
1670         dmu_buf_rele(db, FTAG);
1671         ztest_object_unlock(zd, lr->lr_foid);
1672         return (ENOSPC);
1673     }

1675     bbt = ztest_bt_bonus(db);
1676     ASSERT3U(bbt->bt_magic, ==, BT_MAGIC);
1677     crtngx = bbt->bt_crtxg;
1678     lrtxg = lr->lr_common.lrc_txg;

1680     if (zd->zilog->zl_replay) {
1681         ASSERT(lr->lr_size != 0);
1682         ASSERT(lr->lr_mode != 0);
1683         ASSERT(lrtxg != 0);
1684     } else {
1685         /*
1686          * Randomly change the size and increment the generation.
1687          */
1688         lr->lr_size = (ztest_random(db->db_size / sizeof (*bbt)) + 1) *
1689             sizeof (*bbt);
1690         lr->lr_mode = bbt->bt_gen + 1;
1691         ASSERT(lrtxg == 0);
1692     }

1694     /*
1695      * Verify that the current bonus buffer is not newer than our txg.
1696      */
1697     ztest_bt_verify(bbt, os, lr->lr_foid, -1ULL, lr->lr_mode,
1698         MAX(txg, lrtxg), crtngx);

1700     dmu_buf_will_dirty(db, tx);

1702     ASSERT3U(lr->lr_size, >=, sizeof (*bbt));
1703     ASSERT3U(lr->lr_size, <=, db->db_size);
1704     VERIFY0(dmu_set_bonus(db, lr->lr_size, tx));
1705     VERIFY3U(dmu_set_bonus(db, lr->lr_size, tx), ==, 0);

```

```

1705     bbt = ztest_bt_bonus(db);
1707     ztest_bt_generate(bbt, os, lr->lr_foid, -1ULL, lr->lr_mode, txg, crtngx);
1709     dmu_buf_rele(db, FTAG);
1711     (void) ztest_log_setattr(zd, tx, lr);
1713     dmu_tx_commit(tx);
1715     ztest_object_unlock(zd, lr->lr_foid);
1717     return (0);
1718 }
    _____ unchanged portion omitted _____
1868 /*
1869  * Lookup a bunch of objects. Returns the number of objects not found.
1870  */
1871 static int
1872 ztest_lookup(ztest_ds_t *zd, ztest_od_t *od, int count)
1873 {
1874     int missing = 0;
1875     int error;
1877     ASSERT(_mutex_held(&zd->zd_dirobj_lock));
1879     for (int i = 0; i < count; i++, od++) {
1880         od->od_object = 0;
1881         error = zap_lookup(zd->zd_os, od->od_dir, od->od_name,
1882             sizeof(uint64_t), 1, &od->od_object);
1883         if (error) {
1884             ASSERT(error == ENOENT);
1885             ASSERT(od->od_object == 0);
1886             missing++;
1887         } else {
1888             dmu_buf_t *db;
1889             ztest_block_tag_t *bbt;
1890             dmu_object_info_t doi;
1892             ASSERT(od->od_object != 0);
1893             ASSERT(missing == 0); /* there should be no gaps */
1895             ztest_object_lock(zd, od->od_object, RL_READER);
1896             VERIFY0(dmu_bonus_hold(zd->zd_os,
1897                 VERIFY3U(0, ==, dmu_bonus_hold(zd->zd_os,
1898                     od->od_object, FTAG, &db));
1899             dmu_object_info_from_db(db, &doi);
1900             bbt = ztest_bt_bonus(db);
1901             ASSERT3U(bbt->bt_magic, ==, BT_MAGIC);
1902             od->od_type = doi.doi_type;
1903             od->od_blocksize = doi.doi_data_block_size;
1904             od->od_gen = bbt->bt_gen;
1905             dmu_buf_rele(db, FTAG);
1906             ztest_object_unlock(zd, od->od_object);
1907         }
1909     }
1910     return (missing);
    _____ unchanged portion omitted _____
2238 /*
2239  * Verify that we can't destroy an active pool, create an existing pool,
2240  * or create a pool with a bad vdev spec.
2241  */

```

```

2242 /* ARGSUSED */
2243 void
2244 ztest_spa_create_destroy(ztest_ds_t *zd, uint64_t id)
2245 {
2246     ztest_shared_opts_t *zo = &ztest_opts;
2247     spa_t *spa;
2248     nvlist_t *nvroot;
2250     /*
2251      * Attempt to create using a bad file.
2252      */
2253     nvroot = make_vdev_root("/dev/bogus", NULL, 0, 0, 0, 0, 0, 1);
2254     VERIFY3U(ENOENT, ==,
2255         spa_create("ztest_bad_file", nvroot, NULL, NULL));
2256     nvlist_free(nvroot);
2258     /*
2259      * Attempt to create using a bad mirror.
2260      */
2261     nvroot = make_vdev_root("/dev/bogus", NULL, 0, 0, 0, 0, 2, 1);
2262     VERIFY3U(EEXIST, ==,
2263         spa_create("ztest_bad_mirror", nvroot, NULL, NULL));
2264     nvlist_free(nvroot);
2266     /*
2267      * Attempt to create an existing pool. It shouldn't matter
2268      * what's in the nvroot; we should fail with EEXIST.
2269      */
2270     (void) rw_rdlock(&ztest_name_lock);
2271     nvroot = make_vdev_root("/dev/bogus", NULL, 0, 0, 0, 0, 0, 1);
2272     VERIFY3U(EEXIST, ==, spa_create(zo->zo_pool, nvroot, NULL, NULL));
2273     nvlist_free(nvroot);
2274     VERIFY0(spa_open(zo->zo_pool, &spa, FTAG));
2275     VERIFY3U(0, ==, spa_open(zo->zo_pool, &spa, FTAG));
2276     VERIFY3U(EBUSY, ==, spa_destroy(zo->zo_pool));
2277     spa_close(spa, FTAG);
2279     (void) rw_unlock(&ztest_name_lock);
    _____ unchanged portion omitted _____
3022 /* ARGSUSED */
3023 static int
3024 ztest_objset_destroy_cb(const char *name, void *arg)
3025 {
3026     objset_t *os;
3027     dmu_object_info_t doi;
3028     int error;
3030     /*
3031      * Verify that the dataset contains a directory object.
3032      */
3033     VERIFY0(dmu_objset_hold(name, FTAG, &os));
3034     VERIFY3U(0, ==, dmu_objset_hold(name, FTAG, &os));
3035     error = dmu_object_info(os, ZTEST_DIROBJ, &doi);
3036     if (error != ENOENT) {
3037         /* We could have crashed in the middle of destroying it */
3038         ASSERT0(error);
3039         ASSERT3U(error, ==, 0);
3040         ASSERT3U(doi.doi_type, ==, DMU_OT_ZAP_OTHER);
3041         ASSERT3S(doi.doi_physical_blocks_512, >=, 0);
3042     }
3043     dmu_objset_rele(os, FTAG);
3044     /*
3045      * Destroy the dataset.

```

```

3045     */
3046     VERIFY0(dmu_objset_destroy(name, B_FALSE));
3046     VERIFY3U(0, ==, dmu_objset_destroy(name, B_FALSE));
3047     return (0);
3048 }
    _____
    unchanged_portion_omitted_
    _____

3084 /* ARGSUSED */
3085 void
3086 ztest_dmu_objset_create_destroy(ztest_ds_t *zd, uint64_t id)
3087 {
3088     ztest_ds_t zdtmp;
3089     int iters;
3090     int error;
3091     objset_t *os, *os2;
3092     char name[MAXNAMELEN];
3093     zillog_t *zillog;

3095     (void) rw_rdlock(&ztest_name_lock);

3097     (void) snprintf(name, MAXNAMELEN, "%s/temp_%llu",
3098                    ztest_opts.zo_pool, (u_longlong_t)id);

3100     /*
3101     * If this dataset exists from a previous run, process its replay log
3102     * half of the time.  If we don't replay it, then dmu_objset_destroy()
3103     * (invoked from ztest_objset_destroy_cb()) should just throw it away.
3104     */
3105     if (ztest_random(2) == 0 &&
3106         dmu_objset_own(name, DMU_OST_OTHER, B_FALSE, FTAG, &os) == 0) {
3107         ztest_zd_init(&zdtmp, NULL, os);
3108         zil_replay(os, &zdtmp, ztest_replay_vector);
3109         ztest_zd_fini(&zdtmp);
3110         dmu_objset_disown(os, FTAG);
3111     }

3113     /*
3114     * There may be an old instance of the dataset we're about to
3115     * create lying around from a previous run.  If so, destroy it
3116     * and all of its snapshots.
3117     */
3118     (void) dmu_objset_find(name, ztest_objset_destroy_cb, NULL,
3119                          DS_FIND_CHILDREN | DS_FIND_SNAPSHOTS);

3121     /*
3122     * Verify that the destroyed dataset is no longer in the namespace.
3123     */
3124     VERIFY3U(ENOENT, ==, dmu_objset_hold(name, FTAG, &os));

3126     /*
3127     * Verify that we can create a new dataset.
3128     */
3129     error = ztest_dataset_create(name);
3130     if (error) {
3131         if (error == ENOSPC) {
3132             ztest_record_enospc(FTAG);
3133             (void) rw_unlock(&ztest_name_lock);
3134             return;
3135         }
3136         fatal(0, "dmu_objset_create(%s) = %d", name, error);
3137     }

3139     VERIFY0(dmu_objset_own(name, DMU_OST_OTHER, B_FALSE, FTAG, &os));
3139     VERIFY3U(0, ==,
3140             dmu_objset_own(name, DMU_OST_OTHER, B_FALSE, FTAG, &os));

```

```

3141     ztest_zd_init(&zdtmp, NULL, os);

3143     /*
3144     * Open the intent log for it.
3145     */
3146     zillog = zil_open(os, ztest_get_data);

3148     /*
3149     * Put some objects in there, do a little I/O to them,
3150     * and randomly take a couple of snapshots along the way.
3151     */
3152     iters = ztest_random(5);
3153     for (int i = 0; i < iters; i++) {
3154         ztest_dmu_object_alloc_free(&zdtmp, id);
3155         if (ztest_random(iters) == 0)
3156             (void) ztest_snapshot_create(name, i);
3157     }

3159     /*
3160     * Verify that we cannot create an existing dataset.
3161     */
3162     VERIFY3U(EXIST, ==,
3163             dmu_objset_create(name, DMU_OST_OTHER, 0, NULL, NULL));

3165     /*
3166     * Verify that we can hold an objset that is also owned.
3167     */
3168     VERIFY0(dmu_objset_hold(name, FTAG, &os2));
3169     VERIFY3U(0, ==, dmu_objset_hold(name, FTAG, &os2));
3169     dmu_objset_rele(os2, FTAG);

3171     /*
3172     * Verify that we cannot own an objset that is already owned.
3173     */
3174     VERIFY3U(EBUSY, ==,
3175             dmu_objset_own(name, DMU_OST_OTHER, B_FALSE, FTAG, &os2));

3177     zil_close(zillog);
3178     dmu_objset_disown(os, FTAG);
3179     ztest_zd_fini(&zdtmp);

3181     (void) rw_unlock(&ztest_name_lock);
3182 }
    _____
    unchanged_portion_omitted_
    _____

3352 /*
3353 * Verify that dmu_{read,write} work as expected.
3354 */
3355 void
3356 ztest_dmu_read_write(ztest_ds_t *zd, uint64_t id)
3357 {
3358     objset_t *os = zd->zdtmp;
3359     ztest_od_t od[2];
3360     dmu_tx_t *tx;
3361     int i, freeit, error;
3362     uint64_t n, s, txg;
3363     bufwadm_t *packbuf, *bigbuf, *pack, *bigH, *bigT;
3364     uint64_t packobj, packoff, packsize, bigobj, bigoff, bigsize;
3365     uint64_t chunksize = (1000 + ztest_random(1000)) * sizeof (uint64_t);
3366     uint64_t regions = 997;
3367     uint64_t stride = 123456789ULL;
3368     uint64_t width = 40;
3369     int free_percent = 5;

3371     /*
3372     * This test uses two objects, packobj and bigobj, that are always

```

```

3373     * updated together (i.e. in the same tx) so that their contents are
3374     * in sync and can be compared. Their contents relate to each other
3375     * in a simple way: packobj is a dense array of 'bufwad' structures,
3376     * while bigobj is a sparse array of the same bufwads. Specifically,
3377     * for any index n, there are three bufwads that should be identical:
3378     *
3379     *     packobj, at offset n * sizeof (bufwad_t)
3380     *     bigobj, at the head of the nth chunk
3381     *     bigobj, at the tail of the nth chunk
3382     *
3383     * The chunk size is arbitrary. It doesn't have to be a power of two,
3384     * and it doesn't have any relation to the object blocksize.
3385     * The only requirement is that it can hold at least two bufwads.
3386     *
3387     * Normally, we write the bufwad to each of these locations.
3388     * However, free_percent of the time we instead write zeroes to
3389     * packobj and perform a dmu_free_range() on bigobj. By comparing
3390     * bigobj to packobj, we can verify that the DMU is correctly
3391     * tracking which parts of an object are allocated and free,
3392     * and that the contents of the allocated blocks are correct.
3393     */
3395 /*
3396  * Read the directory info. If it's the first time, set things up.
3397  */
3398 ztest_od_init(&od[0], id, FTAG, 0, DMU_OT_UINT64_OTHER, 0, chunksize);
3399 ztest_od_init(&od[1], id, FTAG, 1, DMU_OT_UINT64_OTHER, 0, chunksize);
3401
3402 if (ztest_object_init(zd, od, sizeof (od), B_FALSE) != 0)
3403     return;
3404
3405 bigobj = od[0].od_object;
3406 packobj = od[1].od_object;
3407 chunksize = od[0].od_gen;
3408 ASSERT(chunksize == od[1].od_gen);
3409
3410 /*
3411  * Prefetch a random chunk of the big object.
3412  * Our aim here is to get some async reads in flight
3413  * for blocks that we may free below; the DMU should
3414  * handle this race correctly.
3415  */
3416 n = ztest_random(regions) * stride + ztest_random(width);
3417 s = 1 + ztest_random(2 * width - 1);
3418 dmu_prefetch(os, bigobj, n * chunksize, s * chunksize);
3419
3420 /*
3421  * Pick a random index and compute the offsets into packobj and bigobj.
3422  */
3423 n = ztest_random(regions) * stride + ztest_random(width);
3424 s = 1 + ztest_random(width - 1);
3425
3426 packoff = n * sizeof (bufwad_t);
3427 packsize = s * sizeof (bufwad_t);
3428
3429 bigoff = n * chunksize;
3430 bigsize = s * chunksize;
3431
3432 packbuf = umem_alloc(packsize, UMEM_NOFAIL);
3433 bigbuf = umem_alloc(bigsize, UMEM_NOFAIL);
3434
3435 /*
3436  * free_percent of the time, free a range of bigobj rather than
3437  * overwriting it.
3438  */
3439 freeit = (ztest_random(100) < free_percent);

```

```

3440     /*
3441     * Read the current contents of our objects.
3442     */
3443     error = dmu_read(os, packobj, packoff, packsize, packbuf,
3444                 DMU_READ_PREFETCH);
3445     ASSERT0(error);
3446     ASSERT3U(error, ==, 0);
3447     error = dmu_read(os, bigobj, bigoff, bigsize, bigbuf,
3448                 DMU_READ_PREFETCH);
3449     ASSERT0(error);
3450     ASSERT3U(error, ==, 0);
3451
3452     /*
3453     * Get a tx for the mods to both packobj and bigobj.
3454     */
3455     tx = dmu_tx_create(os);
3456
3457     dmu_tx_hold_write(tx, packobj, packoff, packsize);
3458
3459     if (freeit)
3460         dmu_tx_hold_free(tx, bigobj, bigoff, bigsize);
3461     else
3462         dmu_tx_hold_write(tx, bigobj, bigoff, bigsize);
3463
3464     txg = ztest_tx_assign(tx, TXG_MIGHTWAIT, FTAG);
3465     if (txg == 0) {
3466         umem_free(packbuf, packsize);
3467         umem_free(bigbuf, bigsize);
3468         return;
3469     }
3470
3471     dmu_object_set_checksum(os, bigobj,
3472                           (enum zio_checksum)ztest_random_dsl_prop(ZFS_PROP_CHECKSUM), tx);
3473
3474     dmu_object_set_compress(os, bigobj,
3475                             (enum zio_compress)ztest_random_dsl_prop(ZFS_PROP_COMPRESSION), tx);
3476
3477     /*
3478     * For each index from n to n + s, verify that the existing bufwad
3479     * in packobj matches the bufwads at the head and tail of the
3480     * corresponding chunk in bigobj. Then update all three bufwads
3481     * with the new values we want to write out.
3482     */
3483     for (i = 0; i < s; i++) {
3484         /* LINTED */
3485         pack = (bufwad_t *)((char *)packbuf + i * sizeof (bufwad_t));
3486         /* LINTED */
3487         bigH = (bufwad_t *)((char *)bigbuf + i * chunksize);
3488         /* LINTED */
3489         bigT = (bufwad_t *)((char *)bigH + chunksize) - 1;
3490
3491         ASSERT((uintptr_t)bigH - (uintptr_t)bigbuf < bigsize);
3492         ASSERT((uintptr_t)bigT - (uintptr_t)bigbuf < bigsize);
3493
3494         if (pack->bw_txg > txg)
3495             fatal(0, "future leak: got %llx, open txg is %llx",
3496                 pack->bw_txg, txg);
3497
3498         if (pack->bw_data != 0 && pack->bw_index != n + i)
3499             fatal(0, "wrong index: got %llx, wanted %llx+%llx",
3500                 pack->bw_index, n, i);
3501
3502         if (bcmp(pack, bigH, sizeof (bufwad_t)) != 0)
3503             fatal(0, "pack/bigH mismatch in %p/%p", pack, bigH);

```

```

3503     if (bcmp(pack, bigT, sizeof (bufwad_t)) != 0)
3504         fatal(0, "pack/bigT mismatch in %p/%p", pack, bigT);

3506     if (freeit) {
3507         bzero(pack, sizeof (bufwad_t));
3508     } else {
3509         pack->bw_index = n + i;
3510         pack->bw_txcg = txg;
3511         pack->bw_data = 1 + ztest_random(-2ULL);
3512     }
3513     *bigH = *pack;
3514     *bigT = *pack;
3515 }

3517 /*
3518  * We've verified all the old bufwads, and made new ones.
3519  * Now write them out.
3520  */
3521 dmufree(os, packobj, packoff, packsize, packbuf, tx);

3523 if (freeit) {
3524     if (ztest_opts.zo_verbose >= 7) {
3525         (void) printf("freeing offset %llx size %llx"
3526             " txg %llx\n",
3527             (u_longlong_t)bigoff,
3528             (u_longlong_t)bigsize,
3529             (u_longlong_t)txg);
3530     }
3531     VERIFY(0 == dmufree_range(os, bigobj, bigoff, bigsize, tx));
3532 } else {
3533     if (ztest_opts.zo_verbose >= 7) {
3534         (void) printf("writing offset %llx size %llx"
3535             " txg %llx\n",
3536             (u_longlong_t)bigoff,
3537             (u_longlong_t)bigsize,
3538             (u_longlong_t)txg);
3539     }
3540     dmufree(os, bigobj, bigoff, bigsize, bigbuf, tx);
3541 }

3543 dmufree(tx);

3545 /*
3546  * Sanity check the stuff we just wrote.
3547  */
3548 {
3549     void *packcheck = umem_alloc(packsize, UMEM_NOFAIL);
3550     void *bigcheck = umem_alloc(bigsize, UMEM_NOFAIL);

3552     VERIFY(0 == dmufree(os, packobj, packoff,
3553         packsize, packcheck, DMU_READ_PREFETCH));
3554     VERIFY(0 == dmufree(os, bigobj, bigoff,
3555         bigsize, bigcheck, DMU_READ_PREFETCH));

3557     ASSERT(bcmp(packbuf, packcheck, packsize) == 0);
3558     ASSERT(bcmp(bigbuf, bigcheck, bigsize) == 0);

3560     umem_free(packcheck, packsize);
3561     umem_free(bigcheck, bigsize);
3562 }

3564     umem_free(packbuf, packsize);
3565     umem_free(bigbuf, bigsize);
3566 }

```

unchanged portion omitted

```

3617 void
3618 ztest_dmufree_read_write_zcopy(ztest_ds_t *zd, uint64_t id)
3619 {
3620     objset_t *os = zd->zd_os;
3621     ztest_od_t od[2];
3622     dmufree_tx_t *tx;
3623     uint64_t i;
3624     int error;
3625     uint64_t n, s, txg;
3626     bufwad_t *packbuf, *bigbuf;
3627     uint64_t packobj, packoff, packsize, bigobj, bigoff, bigsize;
3628     uint64_t blocksize = ztest_random_blocksize();
3629     uint64_t chunksize = blocksize;
3630     uint64_t regions = 997;
3631     uint64_t stride = 123456789ULL;
3632     uint64_t width = 9;
3633     dmufree_buf_t *bonus_db;
3634     arc_buf_t **bigbuf_arcbufs;
3635     dmufree_object_info_t doi;

3637     /*
3638      * This test uses two objects, packobj and bigobj, that are always
3639      * updated together (i.e. in the same tx) so that their contents are
3640      * in sync and can be compared. Their contents relate to each other
3641      * in a simple way: packobj is a dense array of 'bufwad' structures,
3642      * while bigobj is a sparse array of the same bufwads. Specifically,
3643      * for any index n, there are three bufwads that should be identical:
3644      *
3645      *     packobj, at offset n * sizeof (bufwad_t)
3646      *     bigobj, at the head of the nth chunk
3647      *     bigobj, at the tail of the nth chunk
3648      *
3649      * The chunk size is set equal to bigobj block size so that
3650      * dmufree_assign_arcbuf() can be tested for object updates.
3651      */

3653     /*
3654      * Read the directory info. If it's the first time, set things up.
3655      */
3656     ztest_od_init(&od[0], id, FTAG, 0, DMU_OT_UINT64_OTHER, blocksize, 0);
3657     ztest_od_init(&od[1], id, FTAG, 1, DMU_OT_UINT64_OTHER, 0, chunksize);

3659     if (ztest_object_init(zd, od, sizeof (od), B_FALSE) != 0)
3660         return;

3662     bigobj = od[0].od_object;
3663     packobj = od[1].od_object;
3664     blocksize = od[0].od_blocksize;
3665     chunksize = blocksize;
3666     ASSERT(chunksize == od[1].od_gen);

3668     VERIFY(dmufree_object_info(os, bigobj, &doi) == 0);
3669     VERIFY(ISP2(doi.doi_data_block_size));
3670     VERIFY(chunksize == doi.doi_data_block_size);
3671     VERIFY(chunksize >= 2 * sizeof (bufwad_t));

3673     /*
3674      * Pick a random index and compute the offsets into packobj and bigobj.
3675      */
3676     n = ztest_random(regions) * stride + ztest_random(width);
3677     s = 1 + ztest_random(width - 1);

3679     packoff = n * sizeof (bufwad_t);
3680     packsize = s * sizeof (bufwad_t);

3682     bigoff = n * chunksize;

```

```

3683     bigsize = s * chunksize;
3685     packbuf = umem_zalloc(packsize, UMEM_NOFAIL);
3686     bigbuf = umem_zalloc(bigsize, UMEM_NOFAIL);
3688     VERIFY0(dmu_bonus_hold(os, bigobj, FTAG, &bonus_db));
3689     VERIFY3U(0, ==, dmu_bonus_hold(os, bigobj, FTAG, &bonus_db));
3690     bigbuf_arcbufs = umem_zalloc(2 * s * sizeof (arc_buf_t *), UMEM_NOFAIL);
3692     /*
3693     * Iteration 0 test zcopy for DB_UNCACHED dbufs.
3694     * Iteration 1 test zcopy to already referenced dbufs.
3695     * Iteration 2 test zcopy to dirty dbuf in the same txg.
3696     * Iteration 3 test zcopy to dbuf dirty in previous txg.
3697     * Iteration 4 test zcopy when dbuf is no longer dirty.
3698     * Iteration 5 test zcopy when it can't be done.
3699     * Iteration 6 one more zcopy write.
3700     */
3701     for (i = 0; i < 7; i++) {
3702         uint64_t j;
3703         uint64_t off;
3705         /*
3706         * In iteration 5 (i == 5) use arcbufs
3707         * that don't match bigobj blksz to test
3708         * dmu_assign_arcbuf() when it can't directly
3709         * assign an arcbuf to a dbuf.
3710         */
3711         for (j = 0; j < s; j++) {
3712             if (i != 5) {
3713                 bigbuf_arcbufs[j] =
3714                     dmu_request_arcbuf(bonus_db, chunksize);
3715             } else {
3716                 bigbuf_arcbufs[2 * j] =
3717                     dmu_request_arcbuf(bonus_db, chunksize / 2);
3718                 bigbuf_arcbufs[2 * j + 1] =
3719                     dmu_request_arcbuf(bonus_db, chunksize / 2);
3720             }
3721         }
3723         /*
3724         * Get a tx for the mods to both packobj and bigobj.
3725         */
3726         tx = dmu_tx_create(os);
3728         dmu_tx_hold_write(tx, packobj, packoff, packsize);
3729         dmu_tx_hold_write(tx, bigobj, bigoff, bigsize);
3731         txg = ztest_tx_assign(tx, TXG_MIGHTWAIT, FTAG);
3732         if (txg == 0) {
3733             umem_free(packbuf, packsize);
3734             umem_free(bigbuf, bigsize);
3735             for (j = 0; j < s; j++) {
3736                 if (i != 5) {
3737                     dmu_return_arcbuf(bigbuf_arcbufs[j]);
3738                 } else {
3739                     dmu_return_arcbuf(
3740                         bigbuf_arcbufs[2 * j]);
3741                     dmu_return_arcbuf(
3742                         bigbuf_arcbufs[2 * j + 1]);
3743                 }
3744             }
3745             umem_free(bigbuf_arcbufs, 2 * s * sizeof (arc_buf_t *));
3746             dmu_buf_rele(bonus_db, FTAG);
3747             return;

```

```

3748     }
3750     /*
3751     * 50% of the time don't read objects in the 1st iteration to
3752     * test dmu_assign_arcbuf() for the case when there're no
3753     * existing dbufs for the specified offsets.
3754     */
3755     if (i != 0 || ztest_random(2) != 0) {
3756         error = dmu_read(os, packobj, packoff,
3757             packsize, packbuf, DMU_READ_PREFETCH);
3758         ASSERT0(error);
3759         ASSERT3U(error, ==, 0);
3759         error = dmu_read(os, bigobj, bigoff, bigsize,
3760             bigbuf, DMU_READ_PREFETCH);
3761         ASSERT0(error);
3762         ASSERT3U(error, ==, 0);
3763     }
3763     compare_and_update_pbbufs(s, packbuf, bigbuf, bigsize,
3764         n, chunksize, txg);
3766     /*
3767     * We've verified all the old bufwads, and made new ones.
3768     * Now write them out.
3769     */
3770     dmu_write(os, packobj, packoff, packsize, packbuf, tx);
3771     if (ztest_opts.zo_verbose >= 7) {
3772         (void) printf("writing offset %llx size %llx"
3773             " txg %llx\n",
3774             (u_longlong_t)bigoff,
3775             (u_longlong_t)bigsize,
3776             (u_longlong_t)txg);
3777     }
3778     for (off = bigoff, j = 0; j < s; j++, off += chunksize) {
3779         dmu_buf_t *dbt;
3780         if (i != 5) {
3781             bcopy((caddr_t)bigbuf + (off - bigoff),
3782                 bigbuf_arcbufs[j]->b_data, chunksize);
3783         } else {
3784             bcopy((caddr_t)bigbuf + (off - bigoff),
3785                 bigbuf_arcbufs[2 * j]->b_data,
3786                 chunksize / 2);
3787             bcopy((caddr_t)bigbuf + (off - bigoff) +
3788                 chunksize / 2,
3789                 bigbuf_arcbufs[2 * j + 1]->b_data,
3790                 chunksize / 2);
3791         }
3793         if (i == 1) {
3794             VERIFY(dmu_buf_hold(os, bigobj, off,
3795                 FTAG, &dbt, DMU_READ_NO_PREFETCH) == 0);
3796         }
3797         if (i != 5) {
3798             dmu_assign_arcbuf(bonus_db, off,
3799                 bigbuf_arcbufs[j], tx);
3800         } else {
3801             dmu_assign_arcbuf(bonus_db, off,
3802                 bigbuf_arcbufs[2 * j], tx);
3803             dmu_assign_arcbuf(bonus_db,
3804                 off + chunksize / 2,
3805                 bigbuf_arcbufs[2 * j + 1], tx);
3806         }
3807         if (i == 1) {
3808             dmu_buf_rele(dbt, FTAG);
3809         }
3810     }
3811     dmu_tx_commit(tx);

```



```

3813     /*
3814     * Sanity check the stuff we just wrote.
3815     */
3816     {
3817         void *packcheck = umem_alloc(packsize, UMEM_NOFAIL);
3818         void *bigcheck = umem_alloc(bigsize, UMEM_NOFAIL);
3819
3820         VERIFY(0 == dmuf_read(os, packobj, packoff,
3821             packsize, packcheck, DMU_READ_PREFETCH));
3822         VERIFY(0 == dmuf_read(os, bigobj, bigoff,
3823             bigsize, bigcheck, DMU_READ_PREFETCH));
3824
3825         ASSERT(bcmp(packbuf, packcheck, packsize) == 0);
3826         ASSERT(bcmp(bigbuf, bigcheck, bigsize) == 0);
3827
3828         umem_free(packcheck, packsize);
3829         umem_free(bigcheck, bigsize);
3830     }
3831     if (i == 2) {
3832         txg_wait_open(dmuf_objset_pool(os), 0);
3833     } else if (i == 3) {
3834         txg_wait_synced(dmuf_objset_pool(os), 0);
3835     }
3836 }
3837
3838     dmuf_rele(bonus_db, FTAG);
3839     umem_free(packbuf, packsize);
3840     umem_free(bigbuf, bigsize);
3841     umem_free(bigbuf_arcbufs, 2 * s * sizeof(arc_buf_t *));
3842 }
3843
3844 unchanged portion omitted
3845
3846 /*
3847 * Verify that zap_{create,destroy,add,remove,update} work as expected.
3848 */
3849 #define ZTEST_ZAP_MIN_INTS      1
3850 #define ZTEST_ZAP_MAX_INTS     4
3851 #define ZTEST_ZAP_MAX_PROPS    1000
3852
3853 void
3854 ztest_zap(ztest_ds_t *zd, uint64_t id)
3855 {
3856     objset_t *os = zd->zd_os;
3857     ztest_od_t od[1];
3858     uint64_t object;
3859     uint64_t txg, last_txg;
3860     uint64_t value[ZTEST_ZAP_MAX_INTS];
3861     uint64_t zl_ints, zl_intsize, prop;
3862     int i, ints;
3863     dmuf_tx_t *tx;
3864     char proptest[100], txgname[100];
3865     int error;
3866     char *hc[2] = { "s.acl.h", "s.open.h.hyLZlg" };
3867
3868     ztest_od_init(&od[0], id, FTAG, 0, DMU_OT_ZAP_OTHER, 0, 0);
3869
3870     if (ztest_object_init(zd, od, sizeof(od), !ztest_random(2)) != 0)
3871         return;
3872
3873     object = od[0].od_object;
3874
3875     /*
3876     * Generate a known hash collision, and verify that
3877     * we can lookup and remove both entries.
3878     */

```

```

3879     tx = dmuf_tx_create(os);
3880     dmuf_tx_hold_zap(tx, object, B_TRUE, NULL);
3881     txg = ztest_tx_assign(tx, TXG_MIGHTWAIT, FTAG);
3882     if (txg == 0)
3883         return;
3884     for (i = 0; i < 2; i++) {
3885         value[i] = i;
3886         VERIFY0(zap_add(os, object, hc[i], sizeof(uint64_t),
3887             VERIFY3U(0, ==, zap_add(os, object, hc[i], sizeof(uint64_t),
3888                 1, &value[i], tx));
3889         );
3890     }
3891     for (i = 0; i < 2; i++) {
3892         VERIFY3U(ENXIO, ==, zap_add(os, object, hc[i],
3893             sizeof(uint64_t), 1, &value[i], tx));
3894         VERIFY0(zap_length(os, object, hc[i], &zl_intsize, &zl_ints));
3895         VERIFY3U(0, ==,
3896             zap_length(os, object, hc[i], &zl_intsize, &zl_ints));
3897         ASSERT3U(zl_intsize, ==, sizeof(uint64_t));
3898         ASSERT3U(zl_ints, ==, 1);
3899     }
3900     for (i = 0; i < 2; i++) {
3901         VERIFY0(zap_remove(os, object, hc[i], tx));
3902         VERIFY3U(0, ==, zap_remove(os, object, hc[i], tx));
3903     }
3904     dmuf_tx_commit(tx);
3905
3906     /*
3907     * Generate a buch of random entries.
3908     */
3909     ints = MAX(ZTEST_ZAP_MIN_INTS, object % ZTEST_ZAP_MAX_INTS);
3910
3911     prop = ztest_random(ZTEST_ZAP_MAX_PROPS);
3912     (void) sprintf(proptest, "prop_%llu", (u_longlong_t)prop);
3913     (void) sprintf(txgname, "txg_%llu", (u_longlong_t)prop);
3914     bzero(value, sizeof(value));
3915     last_txg = 0;
3916
3917     /*
3918     * If these zap entries already exist, validate their contents.
3919     */
3920     error = zap_length(os, object, txgname, &zl_intsize, &zl_ints);
3921     if (error == 0) {
3922         ASSERT3U(zl_intsize, ==, sizeof(uint64_t));
3923         ASSERT3U(zl_ints, ==, 1);
3924
3925         VERIFY(zap_lookup(os, object, txgname, zl_intsize,
3926             zl_ints, &last_txg) == 0);
3927
3928         VERIFY(zap_length(os, object, proptest, &zl_intsize,
3929             &zl_ints) == 0);
3930
3931         ASSERT3U(zl_intsize, ==, sizeof(uint64_t));
3932         ASSERT3U(zl_ints, ==, ints);
3933
3934         VERIFY(zap_lookup(os, object, proptest, zl_intsize,
3935             zl_ints, value) == 0);
3936
3937         for (i = 0; i < ints; i++) {
3938             ASSERT3U(value[i], ==, last_txg + object + i);
3939         }
3940     } else {
3941         ASSERT3U(error, ==, ENOENT);
3942     }
3943
3944     /*
3945     * Atomically update two entries in our zap object.

```

```

3995     * The first is named txg_%%llu, and contains the txg
3996     * in which the property was last updated.  The second
3997     * is named prop_%%llu, and the nth element of its value
3998     * should be txg + object + n.
3999     */
4000     tx = dmu_tx_create(os);
4001     dmu_tx_hold_zap(tx, object, B_TRUE, NULL);
4002     txg = ztest_tx_assign(tx, TXG_MIGHTWAIT, FTAG);
4003     if (txg == 0)
4004         return;

4006     if (last_txg > txg)
4007         fatal(0, "zap future leak: old %%llu new %%llu", last_txg, txg);

4009     for (i = 0; i < ints; i++)
4010         value[i] = txg + object + i;

4012     VERIFY0(zap_update(os, object, txgname, sizeof (uint64_t),
4013     VERIFY3U(0, ==, zap_update(os, object, txgname, sizeof (uint64_t),
4014     1, &txg, tx));
4014     VERIFY0(zap_update(os, object, propname, sizeof (uint64_t),
4016     VERIFY3U(0, ==, zap_update(os, object, propname, sizeof (uint64_t),
4015     ints, value, tx));

4017     dmu_tx_commit(tx);

4019     /*
4020     * Remove a random pair of entries.
4021     */
4022     prop = ztest_random(ZTEST_ZAP_MAX_PROPS);
4023     (void) sprintf(propname, "prop_%%llu", (u_longlong_t)prop);
4024     (void) sprintf(txgname, "txg_%%llu", (u_longlong_t)prop);

4026     error = zap_length(os, object, txgname, &zl_intsize, &zl_ints);

4028     if (error == ENOENT)
4029         return;

4031     ASSERT0(error);
4033     ASSERT3U(error, ==, 0);

4033     tx = dmu_tx_create(os);
4034     dmu_tx_hold_zap(tx, object, B_TRUE, NULL);
4035     txg = ztest_tx_assign(tx, TXG_MIGHTWAIT, FTAG);
4036     if (txg == 0)
4037         return;
4038     VERIFY0(zap_remove(os, object, txgname, tx));
4039     VERIFY0(zap_remove(os, object, propname, tx));
4040     VERIFY3U(0, ==, zap_remove(os, object, txgname, tx));
4041     VERIFY3U(0, ==, zap_remove(os, object, propname, tx));
4040     dmu_tx_commit(tx);
4041 }
    unchanged portion omitted

4207 /* This is the actual commit callback function */
4208 static void
4209 ztest_commit_callback(void *arg, int error)
4210 {
4211     ztest_cb_data_t *data = arg;
4212     uint64_t synced_txg;

4214     VERIFY(data != NULL);
4215     VERIFY3S(data->zcd_expected_err, ==, error);
4216     VERIFY(!data->zcd_called);

4218     synced_txg = spa_last_synced_txg(data->zcd_spa);

```

```

4219     if (data->zcd_txg > synced_txg)
4220         fatal(0, "commit callback of txg %% PRIu64 " called prematurely"
4221         ", last synced txg = %% PRIu64 "\n", data->zcd_txg,
4222         synced_txg);

4224     data->zcd_called = B_TRUE;

4226     if (error == ECANCELED) {
4227         ASSERT0(data->zcd_txg);
4229         ASSERT3U(data->zcd_txg, ==, 0);
4228         ASSERT(!data->zcd_added);

4230         /*
4231         * The private callback data should be destroyed here, but
4232         * since we are going to check the zcd_called field after
4233         * dmu_tx_abort(), we will destroy it there.
4234         */
4235         return;
4236     }

4238     /* Was this callback added to the global callback list? */
4239     if (!data->zcd_added)
4240         goto out;

4242     ASSERT3U(data->zcd_txg, !=, 0);

4244     /* Remove our callback from the list */
4245     (void) mutex_lock(&zcl.zcl_callbacks_lock);
4246     list_remove(&zcl.zcl_callbacks, data);
4247     (void) mutex_unlock(&zcl.zcl_callbacks_lock);

4249 out:
4250     umem_free(data, sizeof (ztest_cb_data_t));
4251 }
    unchanged portion omitted

4421 /* ARGSUSED */
4422 void
4423 ztest_spa_prop_get_set(ztest_ds_t *zd, uint64_t id)
4424 {
4425     nvlist_t *props = NULL;

4427     (void) rw_rdlock(&ztest_name_lock);

4429     (void) ztest_spa_prop_set_uint64(ZPOOL_PROP_DEDUPDITTO,
4430     ZIO_DEDUPDITTO_MIN + ztest_random(ZIO_DEDUPDITTO_MIN));

4432     VERIFY0(spa_prop_get(ztest_spa, &props));
4434     VERIFY3U(spa_prop_get(ztest_spa, &props), ==, 0);

4434     if (ztest_opts.zo_verbose >= 6)
4435         dump_nvlist(props, 4);

4437     nvlist_free(props);

4439     (void) rw_unlock(&ztest_name_lock);
4440 }
    unchanged portion omitted

4868 /*
4869 * Rename the pool to a different name and then rename it back.
4870 */
4871 /* ARGSUSED */
4872 void
4873 ztest_spa_rename(ztest_ds_t *zd, uint64_t id)
4874 {

```

```

4875     char *oldname, *newname;
4876     spa_t *spa;

4878     (void) rw_wrlock(&ztest_name_lock);

4880     oldname = ztest_opts.zo_pool;
4881     newname = umem_alloc(strlen(oldname) + 5, UMEM_NOFAIL);
4882     (void) strcpy(newname, oldname);
4883     (void) strcat(newname, "_tmp");

4885     /*
4886     * Do the rename
4887     */
4888     VERIFY0(spa_rename(oldname, newname));
4889     VERIFY3U(0, ==, spa_rename(oldname, newname));

4890     /*
4891     * Try to open it under the old name, which shouldn't exist
4892     */
4893     VERIFY3U(ENOENT, ==, spa_open(oldname, &spa, FTAG));

4895     /*
4896     * Open it under the new name and make sure it's still the same spa_t.
4897     */
4898     VERIFY0(spa_open(newname, &spa, FTAG));
4899     VERIFY3U(0, ==, spa_open(newname, &spa, FTAG));

4900     ASSERT(spa == ztest_spa);
4901     spa_close(spa, FTAG);

4903     /*
4904     * Rename it back to the original
4905     */
4906     VERIFY0(spa_rename(newname, oldname));
4907     VERIFY3U(0, ==, spa_rename(newname, oldname));

4908     /*
4909     * Make sure it can still be opened
4910     */
4911     VERIFY0(spa_open(oldname, &spa, FTAG));
4912     VERIFY3U(0, ==, spa_open(oldname, &spa, FTAG));

4913     ASSERT(spa == ztest_spa);
4914     spa_close(spa, FTAG);

4916     umem_free(newname, strlen(newname) + 1);

4918     (void) rw_unlock(&ztest_name_lock);
4919 }
    unchanged portion omitted

4991 static void
4992 ztest_spa_import_export(char *oldname, char *newname)
4993 {
4994     nvlist_t *config, *newconfig;
4995     uint64_t pool_guid;
4996     spa_t *spa;

4998     if (ztest_opts.zo_verbose >= 4) {
4999         (void) printf("import/export: old = %s, new = %s\n",
5000                     oldname, newname);
5001     }

5003     /*
5004     * Clean up from previous runs.
5005     */

```

```

5006     (void) spa_destroy(newname);

5008     /*
5009     * Get the pool's configuration and guid.
5010     */
5011     VERIFY0(spa_open(oldname, &spa, FTAG));
5012     VERIFY3U(0, ==, spa_open(oldname, &spa, FTAG));

5013     /*
5014     * Kick off a scrub to tickle scrub/export races.
5015     */
5016     if (ztest_random(2) == 0)
5017         (void) spa_scan(spa, POOL_SCAN_SCRUB);

5019     pool_guid = spa_guid(spa);
5020     spa_close(spa, FTAG);

5022     ztest_walk_pool_directory("pools before export");

5024     /*
5025     * Export it.
5026     */
5027     VERIFY0(spa_export(oldname, &config, B_FALSE, B_FALSE));
5028     VERIFY3U(0, ==, spa_export(oldname, &config, B_FALSE, B_FALSE));

5029     ztest_walk_pool_directory("pools after export");

5031     /*
5032     * Try to import it.
5033     */
5034     newconfig = spa_tryimport(config);
5035     ASSERT(newconfig != NULL);
5036     nvlist_free(newconfig);

5038     /*
5039     * Import it under the new name.
5040     */
5041     VERIFY0(spa_import(newname, config, NULL, 0));
5042     VERIFY3U(0, ==, spa_import(newname, config, NULL, 0));

5043     ztest_walk_pool_directory("pools after import");

5045     /*
5046     * Try to import it again -- should fail with EEXIST.
5047     */
5048     VERIFY3U(EEXIST, ==, spa_import(newname, config, NULL, 0));

5050     /*
5051     * Try to import it under a different name -- should fail with EEXIST.
5052     */
5053     VERIFY3U(EEXIST, ==, spa_import(oldname, config, NULL, 0));

5055     /*
5056     * Verify that the pool is no longer visible under the old name.
5057     */
5058     VERIFY3U(ENOENT, ==, spa_open(oldname, &spa, FTAG));

5060     /*
5061     * Verify that we can open and close the pool using the new name.
5062     */
5063     VERIFY0(spa_open(newname, &spa, FTAG));
5064     VERIFY3U(0, ==, spa_open(newname, &spa, FTAG));
5065     ASSERT(pool_guid == spa_guid(spa));
5066     spa_close(spa, FTAG);

5067     nvlist_free(config);

```

```

5068 }
    unchanged_portion_omitted
5205 static void
5206 ztest_dataset_dirobj_verify(ztest_ds_t *zd)
5207 {
5208     uint64_t usedobjs, dirobjs, scratch;

5210     /*
5211     * ZTEST_DIROBJ is the object directory for the entire dataset.
5212     * Therefore, the number of objects in use should equal the
5213     * number of ZTEST_DIROBJ entries, +1 for ZTEST_DIROBJ itself.
5214     * If not, we have an object leak.
5215     *
5216     * Note that we can only check this in ztest_dataset_open(),
5217     * when the open-context and syncing-context values agree.
5218     * That's because zap_count() returns the open-context value,
5219     * while dmu_objset_space() returns the rootbp fill count.
5220     */
5221     VERIFY0(zap_count(zd->zd_os, ZTEST_DIROBJ, &dirobjs));
5222     VERIFY3U(0, ==, zap_count(zd->zd_os, ZTEST_DIROBJ, &dirobjs));
5223     dmu_objset_space(zd->zd_os, &scratch, &scratch, &usedobjs, &scratch);
5224     ASSERT3U(dirobjs + 1, ==, usedobjs);
5225 }

5226 static int
5227 ztest_dataset_open(int d)
5228 {
5229     ztest_ds_t *zd = &ztest_ds[d];
5230     uint64_t committed_seq = ZTEST_GET_SHARED_DS(d)->zd_seq;
5231     objset_t *os;
5232     zillog_t *zillog;
5233     char name[MAXNAMELEN];
5234     int error;

5236     ztest_dataset_name(name, ztest_opts.zo_pool, d);

5238     (void) rw_rdlock(&ztest_name_lock);

5240     error = ztest_dataset_create(name);
5241     if (error == ENOSPC) {
5242         (void) rw_unlock(&ztest_name_lock);
5243         ztest_record_enospc(FTAG);
5244         return (error);
5245     }
5246     ASSERT(error == 0 || error == EEXIST);

5248     VERIFY0(dmu_objset_hold(name, zd, &os));
5249     VERIFY3U(dmu_objset_hold(name, zd, &os), ==, 0);
5250     (void) rw_unlock(&ztest_name_lock);

5251     ztest_zd_init(zd, ZTEST_GET_SHARED_DS(d), os);

5253     zillog = zd->zd_zillog;

5255     if (zillog->zl_header->zh_claim_lr_seq != 0 &&
5256         zillog->zl_header->zh_claim_lr_seq < committed_seq)
5257         fatal(0, "missing log records: claimed %llu < committed %llu",
5258             zillog->zl_header->zh_claim_lr_seq, committed_seq);

5260     ztest_dataset_dirobj_verify(zd);

5262     zil_replay(os, zd, ztest_replay_vector);

5264     ztest_dataset_dirobj_verify(zd);

```

```

5266     if (ztest_opts.zo_verbose >= 6)
5267         (void) printf("%s replay %llu blocks, %llu records, seq %llu\n",
5268             zd->zd_name,
5269             (u_longlong_t)zillog->zl_parse_blk_count,
5270             (u_longlong_t)zillog->zl_parse_lr_count,
5271             (u_longlong_t)zillog->zl_replaying_seq);

5273     zillog = zil_open(os, ztest_get_data);

5275     if (zillog->zl_replaying_seq != 0 &&
5276         zillog->zl_replaying_seq < committed_seq)
5277         fatal(0, "missing log records: replayed %llu < committed %llu",
5278             zillog->zl_replaying_seq, committed_seq);

5280     return (0);
5281 }
    unchanged_portion_omitted
5294 /*
5295  * Kick off threads to run tests on all datasets in parallel.
5296  */
5297 static void
5298 ztest_run(ztest_shared_t *zs)
5299 {
5300     thread_t *tid;
5301     spa_t *spa;
5302     objset_t *os;
5303     thread_t resume_tid;
5304     int error;

5306     ztest_exiting = B_FALSE;

5308     /*
5309     * Initialize parent/child shared state.
5310     */
5311     VERIFY(_mutex_init(&ztest_vdev_lock, USYNC_THREAD, NULL) == 0);
5312     VERIFY(rwlock_init(&ztest_name_lock, USYNC_THREAD, NULL) == 0);

5314     zs->zs_thread_start = gethrtime();
5315     zs->zs_thread_stop =
5316         zs->zs_thread_start + ztest_opts.zo_passtime * NANOSEC;
5317     zs->zs_thread_stop = MIN(zs->zs_thread_stop, zs->zs_proc_stop);
5318     zs->zs_thread_kill = zs->zs_thread_stop;
5319     if (ztest_random(100) < ztest_opts.zo_killrate) {
5320         zs->zs_thread_kill -=
5321             ztest_random(ztest_opts.zo_passtime * NANOSEC);
5322     }

5324     (void) _mutex_init(&zcl.zcl_callbacks_lock, USYNC_THREAD, NULL);

5326     list_create(&zcl.zcl_callbacks, sizeof (ztest_cb_data_t),
5327         offsetof(ztest_cb_data_t, zcd_node));

5329     /*
5330     * Open our pool.
5331     */
5332     kernel_init(FREAD | FWRITE);
5333     VERIFY(spa_open(ztest_opts.zo_pool, &spa, FTAG) == 0);
5334     spa->spa_debug = B_TRUE;
5335     ztest_spa = spa;

5337     VERIFY0(dmu_objset_hold(ztest_opts.zo_pool, FTAG, &os));
5338     VERIFY3U(0, ==, dmu_objset_hold(ztest_opts.zo_pool, FTAG, &os));
5339     zs->zs_guid = dmu_objset_fsid_guid(os);
5340     dmu_objset_rele(os, FTAG);

```

```

5341     spa->spa_dedup_ditto = 2 * ZIO_DEDUPDITTO_MIN;

5343     /*
5344     * We don't expect the pool to suspend unless maxfaults == 0,
5345     * in which case ztest_fault_inject() temporarily takes away
5346     * the only valid replica.
5347     */
5348     if (MAXFAULTS() == 0)
5349         spa->spa_failmode = ZIO_FAILURE_MODE_WAIT;
5350     else
5351         spa->spa_failmode = ZIO_FAILURE_MODE_PANIC;

5353     /*
5354     * Create a thread to periodically resume suspended I/O.
5355     */
5356     VERIFY(thr_create(0, 0, ztest_resume_thread, spa, THR_BOUND,
5357         &resume_tid) == 0);

5359     /*
5360     * Create a deadman thread to abort() if we hang.
5361     */
5362     VERIFY(thr_create(0, 0, ztest_deadman_thread, zs, THR_BOUND,
5363         NULL) == 0);

5365     /*
5366     * Verify that we can safely inquire about about any object,
5367     * whether it's allocated or not. To make it interesting,
5368     * we probe a 5-wide window around each power of two.
5369     * This hits all edge cases, including zero and the max.
5370     */
5371     for (int t = 0; t < 64; t++) {
5372         for (int d = -5; d <= 5; d++) {
5373             error = dm_u_object_info(spa->spa_meta_objset,
5374                 (1ULL << t) + d, NULL);
5375             ASSERT(error == 0 || error == ENOENT ||
5376                 error == EINVAL);
5377         }
5378     }

5380     /*
5381     * If we got any ENOSPC errors on the previous run, destroy something.
5382     */
5383     if (zs->zs_enospc_count != 0) {
5384         int d = ztest_random(ztest_opts.zo_datasets);
5385         ztest_dataset_destroy(d);
5386     }
5387     zs->zs_enospc_count = 0;

5389     tid = umem_zalloc(ztest_opts.zo_threads * sizeof (thread_t),
5390         UMEM_NOFAIL);

5392     if (ztest_opts.zo_verbose >= 4)
5393         (void) printf("starting main threads...\n");

5395     /*
5396     * Kick off all the tests that run in parallel.
5397     */
5398     for (int t = 0; t < ztest_opts.zo_threads; t++) {
5399         if (t < ztest_opts.zo_datasets &&
5400             ztest_dataset_open(t) != 0)
5401             return;
5402         VERIFY(thr_create(0, 0, ztest_thread, (void *) (uintptr_t)t,
5403             THR_BOUND, &tid[t]) == 0);
5404     }

5406     /*

```

```

5407     * Wait for all of the tests to complete. We go in reverse order
5408     * so we don't close datasets while threads are still using them.
5409     */
5410     for (int t = ztest_opts.zo_threads - 1; t >= 0; t--) {
5411         VERIFY(thr_join(tid[t], NULL, NULL) == 0);
5412         if (t < ztest_opts.zo_datasets)
5413             ztest_dataset_close(t);
5414     }

5416     txg_wait_synced(spa_get_dsl(spa), 0);

5418     zs->zs_alloc = metaslab_class_get_alloc(spa_normal_class(spa));
5419     zs->zs_space = metaslab_class_get_space(spa_normal_class(spa));

5421     umem_free(tid, ztest_opts.zo_threads * sizeof (thread_t));

5423     /* Kill the resume thread */
5424     ztest_exiting = B_TRUE;
5425     VERIFY(thr_join(resume_tid, NULL, NULL) == 0);
5426     ztest_resume(spa);

5428     /*
5429     * Right before closing the pool, kick off a bunch of async I/O;
5430     * spa_close() should wait for it to complete.
5431     */
5432     for (uint64_t object = 1; object < 50; object++)
5433         dm_u_prefetch(spa->spa_meta_objset, object, 0, 1ULL << 20);

5435     spa_close(spa, FTAG);

5437     /*
5438     * Verify that we can loop over all pools.
5439     */
5440     mutex_enter(&spa_namespace_lock);
5441     for (spa = spa_next(NULL); spa != NULL; spa = spa_next(spa))
5442         if (ztest_opts.zo_verbose > 3)
5443             (void) printf("spa_next: found %s\n", spa_name(spa));
5444     mutex_exit(&spa_namespace_lock);

5446     /*
5447     * Verify that we can export the pool and reimport it under a
5448     * different name.
5449     */
5450     if (ztest_random(2) == 0) {
5451         char name[MAXNAMELEN];
5452         (void) snprintf(name, MAXNAMELEN, "%s_import",
5453             ztest_opts.zo_pool);
5454         ztest_spa_import_export(ztest_opts.zo_pool, name);
5455         ztest_spa_import_export(name, ztest_opts.zo_pool);
5456     }

5458     kernel_fini();

5460     list_destroy(&zcl.zcl_callbacks);

5462     (void) _mutex_destroy(&zcl.zcl_callbacks_lock);

5464     (void) rwlock_destroy(&ztest_name_lock);
5465     (void) _mutex_destroy(&ztest_vdev_lock);
5466 }

5468 static void
5469 ztest_freeze(void)
5470 {
5471     ztest_ds_t *zd = &ztest_ds[0];
5472     spa_t *spa;

```

```

5473     int numloops = 0;
5475     if (ztest_opts.zo_verbose >= 3)
5476         (void) printf("testing spa_freeze()...\n");
5478     kernel_init(FREAD | FWRITE);
5479     VERIFY0(spa_open(ztest_opts.zo_pool, &spa, FTAG));
5480     VERIFY0(ztest_dataset_open(0));
5481     VERIFY3U(0, ==, spa_open(ztest_opts.zo_pool, &spa, FTAG));
5482     VERIFY3U(0, ==, ztest_dataset_open(0));
5482     /*
5483      * Force the first log block to be transactionally allocated.
5484      * We have to do this before we freeze the pool -- otherwise
5485      * the log chain won't be anchored.
5486      */
5487     while (BP_IS_HOLE(&zd->zd_zilog->zl_header->zh_log)) {
5488         ztest_dmu_object_alloc_free(zd, 0);
5489         zil_commit(zd->zd_zilog, 0);
5490     }
5492     txg_wait_synced(spa_get_dsl(spa), 0);
5494     /*
5495      * Freeze the pool. This stops spa_sync() from doing anything,
5496      * so that the only way to record changes from now on is the ZIL.
5497      */
5498     spa_freeze(spa);
5500     /*
5501      * Run tests that generate log records but don't alter the pool config
5502      * or depend on DSL sync tasks (snapshots, objset create/destroy, etc).
5503      * We do a txg_wait_synced() after each iteration to force the txg
5504      * to increase well beyond the last synced value in the uberblock.
5505      * The ZIL should be OK with that.
5506      */
5507     while (ztest_random(10) != 0 &&
5508            numloops++ < ztest_opts.zo_maxloops) {
5509         ztest_dmu_write_parallel(zd, 0);
5510         ztest_dmu_object_alloc_free(zd, 0);
5511         txg_wait_synced(spa_get_dsl(spa), 0);
5512     }
5514     /*
5515      * Commit all of the changes we just generated.
5516      */
5517     zil_commit(zd->zd_zilog, 0);
5518     txg_wait_synced(spa_get_dsl(spa), 0);
5520     /*
5521      * Close our dataset and close the pool.
5522      */
5523     ztest_dataset_close(0);
5524     spa_close(spa, FTAG);
5525     kernel_fini();
5527     /*
5528      * Open and close the pool and dataset to induce log replay.
5529      */
5530     kernel_init(FREAD | FWRITE);
5531     VERIFY0(spa_open(ztest_opts.zo_pool, &spa, FTAG));
5532     VERIFY0(ztest_dataset_open(0));
5533     VERIFY3U(0, ==, spa_open(ztest_opts.zo_pool, &spa, FTAG));
5534     VERIFY3U(0, ==, ztest_dataset_open(0));
5533     ztest_dataset_close(0);
5534     spa_close(spa, FTAG);

```

```

5535         kernel_fini();
5536     }
5537     _____ unchanged_portion_omitted _____
5538
5539     /*
5540      * Create a storage pool with the given name and initial vdev size.
5541      * Then test spa_freeze() functionality.
5542      */
5543     static void
5544     ztest_init(ztest_shared_t *zs)
5545     {
5546         spa_t *spa;
5547         nvlist_t *nvroot, *props;
5548
5549         VERIFY(_mutex_init(&ztest_vdev_lock, USYNC_THREAD, NULL) == 0);
5550         VERIFY(rwlock_init(&ztest_name_lock, USYNC_THREAD, NULL) == 0);
5551
5552         kernel_init(FREAD | FWRITE);
5553
5554         /*
5555          * Create the storage pool.
5556          */
5557         (void) spa_destroy(ztest_opts.zo_pool);
5558         ztest_shared->zs_vdev_next_leaf = 0;
5559         zs->zs_splits = 0;
5560         zs->zs_mirrors = ztest_opts.zo_mirrors;
5561         nvroot = make_vdev_root(NULL, NULL, ztest_opts.zo_vdev_size, 0,
5562                                0, ztest_opts.zo_raidz, zs->zs_mirrors, 1);
5563         props = make_random_props();
5564         for (int i = 0; i < SPA_FEATURES; i++) {
5565             char buf[1024];
5566             (void) snprintf(buf, sizeof(buf), "feature%s",
5567                             spa_feature_table[i].fi_uname);
5568             VERIFY0(nvlist_add_uint64(props, buf, 0));
5569             VERIFY3U(0, ==, nvlist_add_uint64(props, buf, 0));
5570         }
5571         VERIFY0(spa_create(ztest_opts.zo_pool, nvroot, props, NULL));
5572         VERIFY3U(0, ==, spa_create(ztest_opts.zo_pool, nvroot, props, NULL));
5573         nvlist_free(nvroot);
5574
5575         VERIFY0(spa_open(ztest_opts.zo_pool, &spa, FTAG));
5576         VERIFY3U(0, ==, spa_open(ztest_opts.zo_pool, &spa, FTAG));
5577         zs->zs metaslab_sz =
5578             1ULL << spa->spa_root_vdev->vdev_child[0]->vdev_ms_shift;
5579
5580         spa_close(spa, FTAG);
5581
5582         kernel_fini();
5583
5584         ztest_run_zdb(ztest_opts.zo_pool);
5585
5586         ztest_freeze();
5587
5588         ztest_run_zdb(ztest_opts.zo_pool);
5589
5590         (void) rwlock_destroy(&ztest_name_lock);
5591         (void) _mutex_destroy(&ztest_vdev_lock);
5592     }
5593     _____ unchanged_portion_omitted _____
5594
5595     static void
5596     setup_hdr(void)
5597     {
5598         int size;
5599         ztest_shared_hdr_t *hdr;

```

```
5663     hdr = (void *)mmap(0, P2ROUNDUP(sizeof (*hdr), getpagesize()),
5664         PROT_READ | PROT_WRITE, MAP_SHARED, ZTEST_FD_DATA, 0);
5665     ASSERT(hdr != MAP_FAILED);

5666     VERIFY0(ftruncate(ZTEST_FD_DATA, sizeof (ztest_shared_hdr_t)));
5667     VERIFY3U(0, ==, ftruncate(ZTEST_FD_DATA, sizeof (ztest_shared_hdr_t)));

5668     hdr->zh_hdr_size = sizeof (ztest_shared_hdr_t);
5669     hdr->zh_opts_size = sizeof (ztest_shared_opts_t);
5670     hdr->zh_size = sizeof (ztest_shared_t);
5671     hdr->zh_stats_size = sizeof (ztest_shared_callstate_t);
5672     hdr->zh_stats_count = ZTEST_FUNCS;
5673     hdr->zh_ds_size = sizeof (ztest_shared_ds_t);
5674     hdr->zh_ds_count = ztest_opts.zo_datasets;

5675     size = shared_data_size(hdr);
5676     VERIFY0(ftruncate(ZTEST_FD_DATA, size));
5677     VERIFY3U(0, ==, ftruncate(ZTEST_FD_DATA, size));

5678     (void) munmap((caddr_t)hdr, P2ROUNDUP(sizeof (*hdr), getpagesize()));
5679 }
_____unchanged_portion_omitted_____
```

```

*****
9708 Thu Jul 26 14:18:59 2012
new/usr/src/common/nvpair/fnvpair.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 */

22 /*
23  * Copyright (c) 2012 by Delphix. All rights reserved.
24 */

26 #include <sys/nvpair.h>
27 #include <sys/kmem.h>
28 #include <sys/debug.h>
29 #ifndef _KERNEL
30 #include <stdlib.h>
31 #endif

33 /*
34  * "Force" nvlist wrapper.
35  *
36  * These functions wrap the nvlist_* functions with assertions that assume
37  * the operation is successful. This allows the caller's code to be much
38  * more readable, especially for the fnvlist_lookup_* and fnvpair_value_*
39  * functions, which can return the requested value (rather than filling in
40  * a pointer).
41  *
42  * These functions use NV_UNIQUE_NAME, encoding NV_ENCODE_NATIVE, and allocate
43  * with KM_SLEEP.
44  *
45  * More wrappers should be added as needed -- for example
46  * nvlist_lookup_*_array and nvpair_value_*_array.
47  */

49 nvlist_t *
50 fnvlist_alloc(void)
51 {
52     nvlist_t *nvl;
53     VERIFY0(nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP));
54     VERIFY3U(nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP), ==, 0);
55     return (nvl);
56 }
    unchanged_portion_omitted_

63 size_t
64 fnvlist_size(nvlist_t *nvl)
65 {

```

```

66     size_t size;
67     VERIFY0(nvlist_size(nvl, &size, NV_ENCODE_NATIVE));
68     VERIFY3U(nvlist_size(nvl, &size, NV_ENCODE_NATIVE), ==, 0);
69     return (size);
70 }
    unchanged_portion_omitted_

95 nvlist_t *
96 fnvlist_unpack(char *buf, size_t buflen)
97 {
98     nvlist_t *rv;
99     VERIFY0(nvlist_unpack(buf, buflen, &rv, KM_SLEEP));
100    VERIFY3U(nvlist_unpack(buf, buflen, &rv, KM_SLEEP), ==, 0);
101    return (rv);
102 }

103 nvlist_t *
104 fnvlist_dup(nvlist_t *nvl)
105 {
106     nvlist_t *rv;
107     VERIFY0(nvlist_dup(nvl, &rv, KM_SLEEP));
108     VERIFY3U(nvlist_dup(nvl, &rv, KM_SLEEP), ==, 0);
109     return (rv);
110 }

111 void
112 fnvlist_merge(nvlist_t *dst, nvlist_t *src)
113 {
114     VERIFY0(nvlist_merge(dst, src, KM_SLEEP));
115     VERIFY3U(nvlist_merge(dst, src, KM_SLEEP), ==, 0);
116 }

117 void
118 fnvlist_add_boolean(nvlist_t *nvl, const char *name)
119 {
120     VERIFY0(nvlist_add_boolean(nvl, name));
121     VERIFY3U(nvlist_add_boolean(nvl, name), ==, 0);
122 }

123 void
124 fnvlist_add_boolean_value(nvlist_t *nvl, const char *name, boolean_t val)
125 {
126     VERIFY0(nvlist_add_boolean_value(nvl, name, val));
127     VERIFY3U(nvlist_add_boolean_value(nvl, name, val), ==, 0);
128 }

129 void
130 fnvlist_add_byte(nvlist_t *nvl, const char *name, uchar_t val)
131 {
132     VERIFY0(nvlist_add_byte(nvl, name, val));
133     VERIFY3U(nvlist_add_byte(nvl, name, val), ==, 0);
134 }

135 void
136 fnvlist_add_int8(nvlist_t *nvl, const char *name, int8_t val)
137 {
138     VERIFY0(nvlist_add_int8(nvl, name, val));
139     VERIFY3U(nvlist_add_int8(nvl, name, val), ==, 0);
140 }

141 void
142 fnvlist_add_uint8(nvlist_t *nvl, const char *name, uint8_t val)
143 {
144     VERIFY0(nvlist_add_uint8(nvl, name, val));
145     VERIFY3U(nvlist_add_uint8(nvl, name, val), ==, 0);
146 }

```



```

147 void
148 fnvlist_add_int16(nvlist_t *nvl, const char *name, int16_t val)
149 {
150     VERIFY0(nvlist_add_int16(nvl, name, val));
150     VERIFY3U(nvlist_add_int16(nvl, name, val), ==, 0);
151 }

153 void
154 fnvlist_add_uint16(nvlist_t *nvl, const char *name, uint16_t val)
155 {
156     VERIFY0(nvlist_add_uint16(nvl, name, val));
156     VERIFY3U(nvlist_add_uint16(nvl, name, val), ==, 0);
157 }

159 void
160 fnvlist_add_int32(nvlist_t *nvl, const char *name, int32_t val)
161 {
162     VERIFY0(nvlist_add_int32(nvl, name, val));
162     VERIFY3U(nvlist_add_int32(nvl, name, val), ==, 0);
163 }

165 void
166 fnvlist_add_uint32(nvlist_t *nvl, const char *name, uint32_t val)
167 {
168     VERIFY0(nvlist_add_uint32(nvl, name, val));
168     VERIFY3U(nvlist_add_uint32(nvl, name, val), ==, 0);
169 }

171 void
172 fnvlist_add_int64(nvlist_t *nvl, const char *name, int64_t val)
173 {
174     VERIFY0(nvlist_add_int64(nvl, name, val));
174     VERIFY3U(nvlist_add_int64(nvl, name, val), ==, 0);
175 }

177 void
178 fnvlist_add_uint64(nvlist_t *nvl, const char *name, uint64_t val)
179 {
180     VERIFY0(nvlist_add_uint64(nvl, name, val));
180     VERIFY3U(nvlist_add_uint64(nvl, name, val), ==, 0);
181 }

183 void
184 fnvlist_add_string(nvlist_t *nvl, const char *name, const char *val)
185 {
186     VERIFY0(nvlist_add_string(nvl, name, val));
186     VERIFY3U(nvlist_add_string(nvl, name, val), ==, 0);
187 }

189 void
190 fnvlist_add_nvlist(nvlist_t *nvl, const char *name, nvlist_t *val)
191 {
192     VERIFY0(nvlist_add_nvlist(nvl, name, val));
192     VERIFY3U(nvlist_add_nvlist(nvl, name, val), ==, 0);
193 }

195 void
196 fnvlist_add_nvpair(nvlist_t *nvl, nvpair_t *pair)
197 {
198     VERIFY0(nvlist_add_nvpair(nvl, pair));
198     VERIFY3U(nvlist_add_nvpair(nvl, pair), ==, 0);
199 }

201 void
202 fnvlist_add_boolean_array(nvlist_t *nvl, const char *name,

```

```

203     boolean_t *val, uint_t n)
204 {
205     VERIFY0(nvlist_add_boolean_array(nvl, name, val, n));
205     VERIFY3U(nvlist_add_boolean_array(nvl, name, val, n), ==, 0);
206 }

208 void
209 fnvlist_add_byte_array(nvlist_t *nvl, const char *name, uchar_t *val, uint_t n)
210 {
211     VERIFY0(nvlist_add_byte_array(nvl, name, val, n));
211     VERIFY3U(nvlist_add_byte_array(nvl, name, val, n), ==, 0);
212 }

214 void
215 fnvlist_add_int8_array(nvlist_t *nvl, const char *name, int8_t *val, uint_t n)
216 {
217     VERIFY0(nvlist_add_int8_array(nvl, name, val, n));
217     VERIFY3U(nvlist_add_int8_array(nvl, name, val, n), ==, 0);
218 }

220 void
221 fnvlist_add_uint8_array(nvlist_t *nvl, const char *name, uint8_t *val, uint_t n)
222 {
223     VERIFY0(nvlist_add_uint8_array(nvl, name, val, n));
223     VERIFY3U(nvlist_add_uint8_array(nvl, name, val, n), ==, 0);
224 }

226 void
227 fnvlist_add_int16_array(nvlist_t *nvl, const char *name, int16_t *val, uint_t n)
228 {
229     VERIFY0(nvlist_add_int16_array(nvl, name, val, n));
229     VERIFY3U(nvlist_add_int16_array(nvl, name, val, n), ==, 0);
230 }

232 void
233 fnvlist_add_uint16_array(nvlist_t *nvl, const char *name,
234     uint16_t *val, uint_t n)
235 {
236     VERIFY0(nvlist_add_uint16_array(nvl, name, val, n));
236     VERIFY3U(nvlist_add_uint16_array(nvl, name, val, n), ==, 0);
237 }

239 void
240 fnvlist_add_int32_array(nvlist_t *nvl, const char *name, int32_t *val, uint_t n)
241 {
242     VERIFY0(nvlist_add_int32_array(nvl, name, val, n));
242     VERIFY3U(nvlist_add_int32_array(nvl, name, val, n), ==, 0);
243 }

245 void
246 fnvlist_add_uint32_array(nvlist_t *nvl, const char *name,
247     uint32_t *val, uint_t n)
248 {
249     VERIFY0(nvlist_add_uint32_array(nvl, name, val, n));
249     VERIFY3U(nvlist_add_uint32_array(nvl, name, val, n), ==, 0);
250 }

252 void
253 fnvlist_add_int64_array(nvlist_t *nvl, const char *name, int64_t *val, uint_t n)
254 {
255     VERIFY0(nvlist_add_int64_array(nvl, name, val, n));
255     VERIFY3U(nvlist_add_int64_array(nvl, name, val, n), ==, 0);
256 }

258 void
259 fnvlist_add_uint64_array(nvlist_t *nvl, const char *name,

```

```

260     uint64_t *val, uint_t n)
261 {
262     VERIFY0(nvlist_add_uint64_array(nvl, name, val, n));
262     VERIFY3U(nvlist_add_uint64_array(nvl, name, val, n), ==, 0);
263 }

265 void
266 fnvlist_add_string_array(nvlist_t *nvl, const char *name,
267     char * const *val, uint_t n)
268 {
269     VERIFY0(nvlist_add_string_array(nvl, name, val, n));
269     VERIFY3U(nvlist_add_string_array(nvl, name, val, n), ==, 0);
270 }

272 void
273 fnvlist_add_nvlist_array(nvlist_t *nvl, const char *name,
274     nvlist_t **val, uint_t n)
275 {
276     VERIFY0(nvlist_add_nvlist_array(nvl, name, val, n));
276     VERIFY3U(nvlist_add_nvlist_array(nvl, name, val, n), ==, 0);
277 }

279 void
280 fnvlist_remove(nvlist_t *nvl, const char *name)
281 {
282     VERIFY0(nvlist_remove_all(nvl, name));
282     VERIFY3U(nvlist_remove_all(nvl, name), ==, 0);
283 }

285 void
286 fnvlist_remove_nvpair(nvlist_t *nvl, nvpair_t *pair)
287 {
288     VERIFY0(nvlist_remove_nvpair(nvl, pair));
288     VERIFY3U(nvlist_remove_nvpair(nvl, pair), ==, 0);
289 }

291 nvpair_t *
292 fnvlist_lookup_nvpair(nvlist_t *nvl, const char *name)
293 {
294     nvpair_t *rv;
295     VERIFY0(nvlist_lookup_nvpair(nvl, name, &rv));
295     VERIFY3U(nvlist_lookup_nvpair(nvl, name, &rv), ==, 0);
296     return (rv);
297 }
    unchanged_portion_omitted

306 boolean_t
307 fnvlist_lookup_boolean_value(nvlist_t *nvl, const char *name)
308 {
309     boolean_t rv;
310     VERIFY0(nvlist_lookup_boolean_value(nvl, name, &rv));
310     VERIFY3U(nvlist_lookup_boolean_value(nvl, name, &rv), ==, 0);
311     return (rv);
312 }

314 uchar_t
315 fnvlist_lookup_byte(nvlist_t *nvl, const char *name)
316 {
317     uchar_t rv;
318     VERIFY0(nvlist_lookup_byte(nvl, name, &rv));
318     VERIFY3U(nvlist_lookup_byte(nvl, name, &rv), ==, 0);
319     return (rv);
320 }

322 int8_t
323 fnvlist_lookup_int8(nvlist_t *nvl, const char *name)

```

```

324 {
325     int8_t rv;
326     VERIFY0(nvlist_lookup_int8(nvl, name, &rv));
326     VERIFY3U(nvlist_lookup_int8(nvl, name, &rv), ==, 0);
327     return (rv);
328 }

330 int16_t
331 fnvlist_lookup_int16(nvlist_t *nvl, const char *name)
332 {
333     int16_t rv;
334     VERIFY0(nvlist_lookup_int16(nvl, name, &rv));
334     VERIFY3U(nvlist_lookup_int16(nvl, name, &rv), ==, 0);
335     return (rv);
336 }

338 int32_t
339 fnvlist_lookup_int32(nvlist_t *nvl, const char *name)
340 {
341     int32_t rv;
342     VERIFY0(nvlist_lookup_int32(nvl, name, &rv));
342     VERIFY3U(nvlist_lookup_int32(nvl, name, &rv), ==, 0);
343     return (rv);
344 }

346 int64_t
347 fnvlist_lookup_int64(nvlist_t *nvl, const char *name)
348 {
349     int64_t rv;
350     VERIFY0(nvlist_lookup_int64(nvl, name, &rv));
350     VERIFY3U(nvlist_lookup_int64(nvl, name, &rv), ==, 0);
351     return (rv);
352 }

354 uint8_t
355 fnvlist_lookup_uint8_t(nvlist_t *nvl, const char *name)
356 {
357     uint8_t rv;
358     VERIFY0(nvlist_lookup_uint8(nvl, name, &rv));
358     VERIFY3U(nvlist_lookup_uint8(nvl, name, &rv), ==, 0);
359     return (rv);
360 }

362 uint16_t
363 fnvlist_lookup_uint16(nvlist_t *nvl, const char *name)
364 {
365     uint16_t rv;
366     VERIFY0(nvlist_lookup_uint16(nvl, name, &rv));
366     VERIFY3U(nvlist_lookup_uint16(nvl, name, &rv), ==, 0);
367     return (rv);
368 }

370 uint32_t
371 fnvlist_lookup_uint32(nvlist_t *nvl, const char *name)
372 {
373     uint32_t rv;
374     VERIFY0(nvlist_lookup_uint32(nvl, name, &rv));
374     VERIFY3U(nvlist_lookup_uint32(nvl, name, &rv), ==, 0);
375     return (rv);
376 }

378 uint64_t
379 fnvlist_lookup_uint64(nvlist_t *nvl, const char *name)
380 {
381     uint64_t rv;
382     VERIFY0(nvlist_lookup_uint64(nvl, name, &rv));

```

```

382     VERIFY3U(nvlist_lookup_uint64(nvl, name, &rv), ==, 0);
383     return (rv);
384 }

386 char *
387 fnvlist_lookup_string(nvlist_t *nvl, const char *name)
388 {
389     char *rv;
390     VERIFY0(nvlist_lookup_string(nvl, name, &rv));
390     VERIFY3U(nvlist_lookup_string(nvl, name, &rv), ==, 0);
391     return (rv);
392 }

394 nvlist_t *
395 fnvlist_lookup_nvlist(nvlist_t *nvl, const char *name)
396 {
397     nvlist_t *rv;
398     VERIFY0(nvlist_lookup_nvlist(nvl, name, &rv));
398     VERIFY3U(nvlist_lookup_nvlist(nvl, name, &rv), ==, 0);
399     return (rv);
400 }

402 boolean_t
403 fnvpair_value_boolean_value(nvpair_t *nvp)
404 {
405     boolean_t rv;
406     VERIFY0(nvpair_value_boolean_value(nvp, &rv));
406     VERIFY3U(nvpair_value_boolean_value(nvp, &rv), ==, 0);
407     return (rv);
408 }

410 uchar_t
411 fnvpair_value_byte(nvpair_t *nvp)
412 {
413     uchar_t rv;
414     VERIFY0(nvpair_value_byte(nvp, &rv));
414     VERIFY3U(nvpair_value_byte(nvp, &rv), ==, 0);
415     return (rv);
416 }

418 int8_t
419 fnvpair_value_int8(nvpair_t *nvp)
420 {
421     int8_t rv;
422     VERIFY0(nvpair_value_int8(nvp, &rv));
422     VERIFY3U(nvpair_value_int8(nvp, &rv), ==, 0);
423     return (rv);
424 }

426 int16_t
427 fnvpair_value_int16(nvpair_t *nvp)
428 {
429     int16_t rv;
430     VERIFY0(nvpair_value_int16(nvp, &rv));
430     VERIFY3U(nvpair_value_int16(nvp, &rv), ==, 0);
431     return (rv);
432 }

434 int32_t
435 fnvpair_value_int32(nvpair_t *nvp)
436 {
437     int32_t rv;
438     VERIFY0(nvpair_value_int32(nvp, &rv));
438     VERIFY3U(nvpair_value_int32(nvp, &rv), ==, 0);
439     return (rv);
440 }

```

```

442 int64_t
443 fnvpair_value_int64(nvpair_t *nvp)
444 {
445     int64_t rv;
446     VERIFY0(nvpair_value_int64(nvp, &rv));
446     VERIFY3U(nvpair_value_int64(nvp, &rv), ==, 0);
447     return (rv);
448 }

450 uint8_t
451 fnvpair_value_uint8_t(nvpair_t *nvp)
452 {
453     uint8_t rv;
454     VERIFY0(nvpair_value_uint8(nvp, &rv));
454     VERIFY3U(nvpair_value_uint8(nvp, &rv), ==, 0);
455     return (rv);
456 }

458 uint16_t
459 fnvpair_value_uint16(nvpair_t *nvp)
460 {
461     uint16_t rv;
462     VERIFY0(nvpair_value_uint16(nvp, &rv));
462     VERIFY3U(nvpair_value_uint16(nvp, &rv), ==, 0);
463     return (rv);
464 }

466 uint32_t
467 fnvpair_value_uint32(nvpair_t *nvp)
468 {
469     uint32_t rv;
470     VERIFY0(nvpair_value_uint32(nvp, &rv));
470     VERIFY3U(nvpair_value_uint32(nvp, &rv), ==, 0);
471     return (rv);
472 }

474 uint64_t
475 fnvpair_value_uint64(nvpair_t *nvp)
476 {
477     uint64_t rv;
478     VERIFY0(nvpair_value_uint64(nvp, &rv));
478     VERIFY3U(nvpair_value_uint64(nvp, &rv), ==, 0);
479     return (rv);
480 }

482 char *
483 fnvpair_value_string(nvpair_t *nvp)
484 {
485     char *rv;
486     VERIFY0(nvpair_value_string(nvp, &rv));
486     VERIFY3U(nvpair_value_string(nvp, &rv), ==, 0);
487     return (rv);
488 }

490 nvlist_t *
491 fnvpair_value_nvlist(nvpair_t *nvp)
492 {
493     nvlist_t *rv;
494     VERIFY0(nvpair_value_nvlist(nvp, &rv));
494     VERIFY3U(nvpair_value_nvlist(nvp, &rv), ==, 0);
495     return (rv);
496 }

```

unchanged portion omitted

new/usr/src/lib/libc/port/threads/assfail.c

1

\*\*\*\*\*

12567 Thu Jul 26 14:19:00 2012

new/usr/src/lib/libc/port/threads/assfail.c

3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero

\*\*\*\*\*

unchanged\_portion\_omitted\_

```
449 void
450 assfail3(const char *assertion, uintmax_t lv, const char *op, uintmax_t rv,
451          const char *filename, int line_num)
452 {
453     char buf[1000];
454     (void) strcpy(buf, assertion);
455     (void) strcat(buf, " (");
456     (void) strcat(buf, " (0x");
457     ultos((uint64_t)lv, 16, buf + strlen(buf));
458     (void) strcat(buf, " ");
459     (void) strcat(buf, op);
460     (void) strcat(buf, " ");
461     ultos((uint64_t)rv, 16, buf + strlen(buf));
462     (void) strcat(buf, ")\n");
463     __assfail(buf, filename, line_num);
464 }
```

unchanged\_portion\_omitted\_

```

*****
2139 Thu Jul 26 14:19:01 2012
new/usr/src/lib/libzpool/Makefile.com
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 LIBRARY= libzpool.a
27 VERS= .1

29 # include the list of ZFS sources
30 include ../../uts/common/Makefile.files
31 KERNEL_OBJS = kernel.o taskq.o util.o

33 OBJECTS=$(ZFS_COMMON_OBJS) $(ZFS_SHARED_OBJS) $(KERNEL_OBJS)

35 # include library definitions
36 include ../../Makefile.lib

38 ZFS_COMMON_SRCS=      $(ZFS_COMMON_OBJS:%.o=../../uts/common/fs/zfs/%.c)
39 ZFS_SHARED_SRCS=     $(ZFS_SHARED_OBJS:%.o=../../common/zfs/%.c)
40 KERNEL_SRCS=         $(KERNEL_OBJS:%.o=../common/%.c)

42 SRCS=$(ZFS_COMMON_SRCS) $(ZFS_SHARED_SRCS) $(KERNEL_SRCS)
43 SRCDIR=      ../common

45 # There should be a mapfile here
46 MAPFILES =

48 LIBS +=      $(LINTLIB)

50 INCS += -I../common
51 INCS += -I../../uts/common/fs/zfs
52 INCS += -I../../common/zfs
53 INCS += -I../../common

55 $(LINTLIB) := SRCS=      $(SRCDIR)/$(LINTSRC)

57 C99MODE=      -xc99=%all
58 C99LMODE=     -Xc99=%all

60 CFLAGS +=     -g $(CCVERBOSE) $(CNOGLOBAL)
61 CFLAGS64 +=  -g $(CCVERBOSE) $(CNOGLOBAL)

```

```

62 LDLIBS +=     -lcmdutils -lumem -lavl -lnvpair -lz -lc -lsysevent -lmd
63 CPPFLAGS +=   $(INCS) -DDEBUG
62 CPPFLAGS +=   $(INCS)

65 .KEEP_STATE:

67 all: $(LIBS)

69 lint: $(LINTLIB)

71 include ../../Makefile.targ

73 pics/%.o: ../../uts/common/fs/zfs/%.c
74     $(COMPILE.c) -o $@ $<
75     $(POST_PROCESS_O)

77 pics/%.o: ../../common/zfs/%.c
78     $(COMPILE.c) -o $@ $<
79     $(POST_PROCESS_O)

```

```

*****
15333 Thu Jul 26 14:19:02 2012
new/usr/src/lib/libzpool/common/sys/zfs_context.h
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 2011 Nexenta Systems, Inc. All rights reserved.
24 * Copyright (c) 2012 by Delphix. All rights reserved.
25 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
26 */

28 #ifndef _SYS_ZFS_CONTEXT_H
29 #define _SYS_ZFS_CONTEXT_H

31 #ifdef __cplusplus
32 extern "C" {
33 #endif

35 #define _SYS_MUTEX_H
36 #define _SYS_RWLOCK_H
37 #define _SYS_CONDVAR_H
38 #define _SYS_SYSTEM_H
39 #define _SYS_DEBUG_H
39 #define _SYS_T_LOCK_H
40 #define _SYS_VNODE_H
41 #define _SYS_VFS_H
42 #define _SYS_SUNDDI_H
43 #define _SYS_CALLB_H

45 #include <stdio.h>
46 #include <stdlib.h>
47 #include <stddef.h>
48 #include <stdarg.h>
49 #include <fcntl.h>
50 #include <unistd.h>
51 #include <errno.h>
52 #include <string.h>
53 #include <strings.h>
54 #include <synch.h>
55 #include <thread.h>
56 #include <assert.h>
57 #include <alloca.h>
58 #include <umem.h>
59 #include <limits.h>
60 #include <atomic.h>

```

```

61 #include <dirent.h>
62 #include <time.h>
63 #include <libsysevent.h>
64 #include <sys/note.h>
65 #include <sys/types.h>
66 #include <sys/cred.h>
67 #include <sys/sysmacros.h>
68 #include <sys/bitmap.h>
69 #include <sys/resource.h>
70 #include <sys/byteorder.h>
71 #include <sys/list.h>
72 #include <sys/uiio.h>
73 #include <sys/zfs_debug.h>
74 #include <sys/sdt.h>
75 #include <sys/kstat.h>
76 #include <sys/u8_textprep.h>
77 #include <sys/sysevent/eventdefs.h>
78 #include <sys/sysevent/dev.h>
79 #include <sys/sunddi.h>
80 #include <sys/debug.h>

82 /*
83  * Debugging
84  */

86 /*
87  * Note that we are not using the debugging levels.
88  */

90 #define CE_CONT          0          /* continuation      */
91 #define CE_NOTE         1          /* notice            */
92 #define CE_WARN        2          /* warning           */
93 #define CE_PANIC       3          /* panic             */
94 #define CE_IGNORE      4          /* print nothing     */

96 /*
97  * ZFS debugging
98  */

100 #ifdef ZFS_DEBUG
101 extern void dprintf_setup(int *argc, char **argv);
102 #endif /* ZFS_DEBUG */

104 extern void cmn_err(int, const char *, ...);
105 extern void vcmn_err(int, const char *, __va_list);
106 extern void panic(const char *, ...);
107 extern void vpanic(const char *, __va_list);

109 #define fm_panic        panic

111 extern int aok;

113 /* This definition is copied from assert.h. */
114 #if defined(__STDC__)
115 #if __STDC_VERSION__ - 0 >= 199901L
116 #define zverify(EX) (void)((EX) || (aok) || \
117     (__assert_c99(#EX, __FILE__, __LINE__, __func__), 0))
118 #else
119 #define zverify(EX) (void)((EX) || (aok) || \
120     (__assert(#EX, __FILE__, __LINE__), 0))
121 #endif /* __STDC_VERSION__ - 0 >= 199901L */
122 #else
123 #define zverify(EX) (void)((EX) || (aok) || \
124     (__assert("EX", __FILE__, __LINE__), 0))
125 #endif /* __STDC__ */

```

```

128 #define VERIFY    zverify
129 #define ASSERT    zverify
130 #undef  assert
131 #define assert    zverify

133 extern void __assert(const char *, const char *, int);

135 #ifdef lint
136 #define VERIFY3_IMPL(x, y, z, t)    if (x == z) ((void)0)
137 #else
138 /* BEGIN CSTYLED */
139 #define VERIFY3_IMPL(LEFT, OP, RIGHT, TYPE) do { \
140     const TYPE __left = (TYPE)(LEFT); \
141     const TYPE __right = (TYPE)(RIGHT); \
142     if (!(__left OP __right) && (!aok)) { \
143         char *__buf = alloca(256); \
144         (void) snprintf(__buf, 256, "%s %s %s (0x%llx %s 0x%llx)", \
145             #LEFT, #OP, #RIGHT, \
146             (u_longlong_t)__left, #OP, (u_longlong_t)__right); \
147         __assert(__buf, __FILE__, __LINE__); \
148     } \
149     _NOTE(CONSTCOND) } while (0)
150 /* END CSTYLED */
151 #endif /* lint */

153 #define VERIFY3S(x, y, z)    VERIFY3_IMPL(x, y, z, int64_t)
154 #define VERIFY3U(x, y, z)    VERIFY3_IMPL(x, y, z, uint64_t)
155 #define VERIFY3P(x, y, z)    VERIFY3_IMPL(x, y, z, uintptr_t)

157 #ifdef NDEEBUG
158 #define ASSERT3S(x, y, z)    ((void)0)
159 #define ASSERT3U(x, y, z)    ((void)0)
160 #define ASSERT3P(x, y, z)    ((void)0)
161 #else
162 #define ASSERT3S(x, y, z)    VERIFY3S(x, y, z)
163 #define ASSERT3U(x, y, z)    VERIFY3U(x, y, z)
164 #define ASSERT3P(x, y, z)    VERIFY3P(x, y, z)
165 #endif

113 /*
114  * DTrace SDT probes have different signatures in userland than they do in
115  * kernel.  If they're being used in kernel code, re-define them out of
116  * existence for their counterparts in libzpool.
117  */

119 #ifdef DTRACE_PROBE
120 #undef  DTRACE_PROBE
121 #define DTRACE_PROBE(a) ((void)0)
122 #endif /* DTRACE_PROBE */

124 #ifdef DTRACE_PROBE1
125 #undef  DTRACE_PROBE1
126 #define DTRACE_PROBE1(a, b, c) ((void)0)
127 #endif /* DTRACE_PROBE1 */

129 #ifdef DTRACE_PROBE2
130 #undef  DTRACE_PROBE2
131 #define DTRACE_PROBE2(a, b, c, d, e) ((void)0)
132 #endif /* DTRACE_PROBE2 */

134 #ifdef DTRACE_PROBE3
135 #undef  DTRACE_PROBE3
136 #define DTRACE_PROBE3(a, b, c, d, e, f, g) ((void)0)
137 #endif /* DTRACE_PROBE3 */

```

```

139 #ifdef DTRACE_PROBE4
140 #undef  DTRACE_PROBE4
141 #define DTRACE_PROBE4(a, b, c, d, e, f, g, h, i) ((void)0)
142 #endif /* DTRACE_PROBE4 */

144 /*
145  * Threads
146  */
147 #define curthread    ((void *) (uintptr_t) thr_self())

149 typedef struct kthread kthread_t;

151 #define thread_create(stk, stksize, func, arg, len, pp, state, pri) \
152     zk_thread_create(func, arg)
153 #define thread_exit()    thr_exit(NULL)
154 #define thread_join(t)    panic("libzpool cannot join threads")

156 #define newproc(f, a, cid, pri, ctp, pid)    (ENOSYS)

158 /* in libzpool, p0 exists only to have its address taken */
159 struct proc {
160     uintptr_t    this_is_never_used_dont_dereference_it;
161 };

```

unchanged portion omitted

```

*****
37877 Thu Jul 26 14:19:03 2012
new/usr/src/uts/common/disp/sysdc.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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) 2009, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 */

26 /*
27 * The System Duty Cycle (SDC) scheduling class
28 * -----
29 *
30 * Background
31 *
32 * Kernel threads in Solaris have traditionally not been large consumers
33 * of CPU time. They typically wake up, perform a small amount of
34 * work, then go back to sleep waiting for either a timeout or another
35 * signal. On the assumption that the small amount of work that they do
36 * is important for the behavior of the whole system, these threads are
37 * treated kindly by the dispatcher and the SYS scheduling class: they run
38 * without preemption from anything other than real-time and interrupt
39 * threads; when preempted, they are put at the front of the queue, so they
40 * generally do not migrate between CPUs; and they are allowed to stay
41 * running until they voluntarily give up the CPU.
42 *
43 * As Solaris has evolved, new workloads have emerged which require the
44 * kernel to perform significant amounts of CPU-intensive work. One
45 * example of such a workload is ZFS's transaction group sync processing.
46 * Each sync operation generates a large batch of I/Os, and each I/O
47 * may need to be compressed and/or checksummed before it is written to
48 * storage. The taskq threads which perform the compression and checksums
49 * will run nonstop as long as they have work to do; a large sync operation
50 * on a compression-heavy dataset can keep them busy for seconds on end.
51 * This causes human-time-scale dispatch latency bubbles for any other
52 * threads which have the misfortune to share a CPU with the taskq threads.
53 *
54 * The SDC scheduling class is a solution to this problem.
55 *
56 *
57 * Overview
58 *
59 * SDC is centered around the concept of a thread's duty cycle (DC):
60 *
61 * ONPROC time

```

```

62 * Duty Cycle = -----
63 * ONPROC + Runnable time
64 *
65 * This is the ratio of the time that the thread spent running on a CPU
66 * divided by the time it spent running or trying to run. It is unaffected
67 * by any time the thread spent sleeping, stopped, etc.
68 *
69 * A thread joining the SDC class specifies a "target" DC that it wants
70 * to run at. To implement this policy, the routine sysdc_update() scans
71 * the list of active SDC threads every few ticks and uses each thread's
72 * microstate data to compute the actual duty cycle that that thread
73 * has experienced recently. If the thread is under its target DC, its
74 * priority is increased to the maximum available (sysdc_maxpri, which is
75 * 99 by default). If the thread is over its target DC, its priority is
76 * reduced to the minimum available (sysdc_minpri, 0 by default). This
77 * is a fairly primitive approach, in that it doesn't use any of the
78 * intermediate priorities, but it's not completely inappropriate. Even
79 * though threads in the SDC class might take a while to do their job, they
80 * are by some definition important if they're running inside the kernel,
81 * so it is reasonable that they should get to run at priority 99.
82 *
83 * If a thread is running when sysdc_update() calculates its actual duty
84 * cycle, and there are other threads of equal or greater priority on its
85 * CPU's dispatch queue, sysdc_update() preempts that thread. The thread
86 * acknowledges the preemption by calling sysdc_preempt(), which calls
87 * setbackdq(), which gives other threads with the same priority a chance
88 * to run. This creates a de facto time quantum for threads in the SDC
89 * scheduling class.
90 *
91 * An SDC thread which is assigned priority 0 can continue to run if
92 * nothing else needs to use the CPU that it's running on. Similarly, an
93 * SDC thread at priority 99 might not get to run as much as it wants to
94 * if there are other priority-99 or higher threads on its CPU. These
95 * situations would cause the thread to get ahead of or behind its target
96 * DC; the longer the situations lasted, the further ahead or behind the
97 * thread would get. Rather than condemning a thread to a lifetime of
98 * paying for its youthful indiscretions, SDC keeps "base" values for
99 * ONPROC and Runnable times in each thread's sysdc data, and updates these
100 * values periodically. The duty cycle is then computed using the elapsed
101 * amount of ONPROC and Runnable times since those base times.
102 *
103 * Since sysdc_update() scans SDC threads fairly frequently, it tries to
104 * keep the list of "active" threads small by pruning out threads which
105 * have been asleep for a brief time. They are not pruned immediately upon
106 * going to sleep, since some threads may bounce back and forth between
107 * sleeping and being runnable.
108 *
109 *
110 * Interfaces
111 *
112 * void sysdc_thread_enter(t, dc, flags)
113 *
114 * Moves a kernel thread from the SYS scheduling class to the
115 * SDC class. t must have an associated LWP (created by calling
116 * lwp_kernel_create()). The thread will have a target DC of dc.
117 * Flags should be either 0 or SYSDC_THREAD_BATCH. If
118 * SYSDC_THREAD_BATCH is specified, the thread is expected to be
119 * doing large amounts of processing.
120 *
121 *
122 * Complications
123 *
124 * - Run queue balancing
125 *
126 * The Solaris dispatcher is biased towards letting a thread run
127 * on the same CPU which it last ran on, if no more than 3 ticks

```



```

128 *      (i.e. rechoose_interval) have passed since the thread last ran.
129 *      This helps to preserve cache warmth.  On the other hand, it also
130 *      tries to keep the per-CPU run queues fairly balanced; if the CPU
131 *      chosen for a runnable thread has a run queue which is three or
132 *      more threads longer than a neighboring CPU's queue, the runnable
133 *      thread is dispatched onto the neighboring CPU instead.
134 *
135 *      These policies work well for some workloads, but not for many SDC
136 *      threads.  The taskq client of SDC, for example, has many discrete
137 *      units of work to do.  The work units are largely independent, so
138 *      cache warmth is not an important consideration.  It is important
139 *      that the threads fan out quickly to different CPUs, since the
140 *      amount of work these threads have to do (a few seconds worth at a
141 *      time) doesn't leave much time to correct thread placement errors
142 *      (i.e. two SDC threads being dispatched to the same CPU).
143 *
144 *      To fix this, SDC uses the TS_RUNQMATCH flag introduced for FSS.
145 *      This tells the dispatcher to keep neighboring run queues' lengths
146 *      more evenly matched, which allows SDC threads to migrate more
147 *      easily.
148 *
149 * - LWPs and system processes
150 *
151 *      SDC can only be used for kernel threads.  Since SDC uses microstate
152 *      accounting data to compute each thread's actual duty cycle, all
153 *      threads entering the SDC class must have associated LWPs (which
154 *      store the microstate data).  This means that the threads have to
155 *      be associated with an SSYS process, i.e. one created by newproc().
156 *      If the microstate accounting information is ever moved into the
157 *      kthread_t, this restriction could be lifted.
158 *
159 * - Dealing with oversubscription
160 *
161 *      Since SDC duty cycles are per-thread, it is possible that the
162 *      aggregate requested duty cycle of all SDC threads in a processor
163 *      set could be greater than the total CPU time available in that set.
164 *      The FSS scheduling class has an analogous situation, which it deals
165 *      with by reducing each thread's allotted CPU time proportionally.
166 *      Since SDC doesn't need to be as precise as FSS, it uses a simpler
167 *      solution to the oversubscription problem.
168 *
169 *      sysdc_update() accumulates the amount of time that max-priority SDC
170 *      threads have spent on-CPU in each processor set, and uses that sum
171 *      to create an implied duty cycle for that processor set:
172 *
173 *          accumulated CPU time
174 *      pset DC = -----
175 *                (# CPUs) * time since last update
176 *
177 *      If this implied duty cycle is above a maximum pset duty cycle (90%
178 *      by default), sysdc_update() sets the priority of all SDC threads
179 *      in that processor set to sysdc_minpri for a "break" period.  After
180 *      the break period, it waits for a "nobreak" period before trying to
181 *      enforce the pset duty cycle limit again.
182 *
183 * - Processor sets
184 *
185 *      As the above implies, SDC is processor set aware, but it does not
186 *      currently allow threads to change processor sets while in the SDC
187 *      class.  Instead, those threads must join the desired processor set
188 *      before entering SDC. [1]
189 *
190 * - Batch threads
191 *
192 *      A thread joining the SDC class can specify the SDC_THREAD_BATCH
193 *      flag.  This flag currently has no effect, but marks threads which

```

```

194 *      do bulk processing.
195 *
196 * - t_kpri_req
197 *
198 *      The TS and FSS scheduling classes pay attention to t_kpri_req,
199 *      which provides a simple form of priority inheritance for
200 *      synchronization primitives (such as rwlocks held as READER) which
201 *      cannot be traced to a unique thread.  The SDC class does not honor
202 *      t_kpri_req, for a few reasons:
203 *
204 *      1. t_kpri_req is notoriously inaccurate.  A measure of its
205 *      inaccuracy is that it needs to be cleared every time a thread
206 *      returns to user mode, because it is frequently non-zero at that
207 *      point.  This can happen because "ownership" of synchronization
208 *      primitives that use t_kpri_req can be silently handed off,
209 *      leaving no opportunity to will the t_kpri_req inheritance.
210 *
211 *      2. Unlike in TS and FSS, threads in SDC *will* eventually run at
212 *      kernel priority.  This means that even if an SDC thread
213 *      is holding a synchronization primitive and running at low
214 *      priority, its priority will eventually be raised above 60,
215 *      allowing it to drive on and release the resource.
216 *
217 *      3. The first consumer of SDC uses the taskq subsystem, which holds
218 *      a reader lock for the duration of the task's execution.  This
219 *      would mean that SDC threads would never drop below kernel
220 *      priority in practice, which defeats one of the purposes of SDC.
221 *
222 * - Why not FSS?
223 *
224 *      It might seem that the existing FSS scheduling class could solve
225 *      the problems that SDC is attempting to solve.  FSS's more precise
226 *      solution to the oversubscription problem would hardly cause
227 *      trouble, as long as it performed well.  SDC is implemented as
228 *      a separate scheduling class for two main reasons: the initial
229 *      consumer of SDC does not map well onto the "project" abstraction
230 *      that is central to FSS, and FSS does not expect to run at kernel
231 *      priorities.
232 *
233 *
234 * Tunables
235 *
236 * - sysdc_update_interval_msec: Number of milliseconds between
237 *   consecutive thread priority updates.
238 *
239 * - sysdc_reset_interval_msec: Number of milliseconds between
240 *   consecutive resets of a thread's base ONPROC and Runnable
241 *   times.
242 *
243 * - sysdc_prune_interval_msec: Number of milliseconds of sleeping
244 *   before a thread is pruned from the active list.
245 *
246 * - sysdc_max_pset_DC: Allowable percentage of a processor set's
247 *   CPU time which SDC can give to its high-priority threads.
248 *
249 * - sysdc_break_msec: Number of milliseconds of "break" taken when
250 *   sysdc_max_pset_DC is exceeded.
251 *
252 *
253 * Future work (in SDC and related subsystems)
254 *
255 * - Per-thread rechoose interval (0 for SDC)
256 *
257 *      Allow each thread to specify its own rechoose interval.  SDC
258 *      threads would specify an interval of zero, which would rechoose
259 *      the CPU with the lowest priority once per update.

```

```

260 *
261 * - Allow threads to change processor sets after joining the SDC class
262 *
263 * - Thread groups and per-group DC
264 *
265 *     It might be nice to be able to specify a duty cycle which applies
266 *     to a group of threads in aggregate.
267 *
268 * - Per-group DC callback to allow dynamic DC tuning
269 *
270 *     Currently, DCs are assigned when the thread joins SDC. Some
271 *     workloads could benefit from being able to tune their DC using
272 *     subsystem-specific knowledge about the workload.
273 *
274 * - Finer-grained priority updates
275 *
276 * - More nuanced management of oversubscription
277 *
278 * - Moving other CPU-intensive threads into SDC
279 *
280 * - Move msacct data into kthread_t
281 *
282 *     This would allow kernel threads without LWPs to join SDC.
283 *
284 *
285 * Footnotes
286 *
287 * [1] The details of doing so are left as an exercise for the reader.
288 */

290 #include <sys/types.h>
291 #include <sys/sysdc.h>
292 #include <sys/sysdc_impl.h>

294 #include <sys/class.h>
295 #include <sys/cmn_err.h>
296 #include <sys/cpuvar.h>
297 #include <sys/cputpart.h>
298 #include <sys/debug.h>
299 #include <sys/disp.h>
300 #include <sys/errno.h>
301 #include <sys/inline.h>
302 #include <sys/kmem.h>
303 #include <sys/modctl.h>
304 #include <sys/schedctl.h>
305 #include <sys/sdt.h>
306 #include <sys/sunddi.h>
307 #include <sys/sysmacros.h>
308 #include <sys/system.h>
309 #include <sys/var.h>

311 /*
312 * Tunables - loaded into the internal state at module load time
313 */
314 uint_t      sysdc_update_interval_msec = 20;
315 uint_t      sysdc_reset_interval_msec = 400;
316 uint_t      sysdc_prune_interval_msec = 100;
317 uint_t      sysdc_max_pset_DC = 90;
318 uint_t      sysdc_break_msec = 80;

320 /*
321 * Internal state - constants set up by sysdc_initparam()
322 */
323 static clock_t sysdc_update_ticks; /* ticks between updates */
324 static uint_t sysdc_prune_updates; /* updates asleep before pruning */
325 static uint_t sysdc_reset_updates; /* # of updates before reset */

```

```

326 static uint_t sysdc_break_updates; /* updates to break */
327 static uint_t sysdc_nobreak_updates; /* updates to not check */
328 static uint_t sysdc_minDC; /* minimum allowed DC */
329 static uint_t sysdc_maxDC; /* maximum allowed DC */
330 static pri_t sysdc_minpri; /* minimum allowed priority */
331 static pri_t sysdc_maxpri; /* maximum allowed priority */

333 /*
334 * Internal state
335 */
336 static kmutex_t sysdc_pset_lock; /* lock protecting pset data */
337 static list_t sysdc_psets; /* list of psets with SDC threads */
338 static uint_t sysdc_param_init; /* sysdc_initparam() has been called */
339 static uint_t sysdc_update_timeout_started; /* update timeout is active */
340 static hrtime_t sysdc_last_update; /* time of last sysdc_update() */
341 static sysdc_t sysdc_dummy; /* used to terminate active lists */

343 /*
344 * Internal state - active hash table
345 */
346 #define SYSDC_NLISTS 8
347 #define SYSDC_HASH(sdc) (((uintptr_t)(sdc) >> 6) & (SYSDC_NLISTS - 1))
348 static sysdc_list_t sysdc_active[SYSDC_NLISTS];
349 #define SYSDC_LIST(sdc) (&sysdc_active[SYSDC_HASH(sdc)])

351 #ifdef DEBUG
352 static struct {
353     uint64_t sysdc_update_times_asleep;
354     uint64_t sysdc_update_times_base_ran_backwards;
355     uint64_t sysdc_update_times_already_done;
356     uint64_t sysdc_update_times_cur_ran_backwards;
357     uint64_t sysdc_compute_pri_breaking;
358     uint64_t sysdc_activate_enter;
359     uint64_t sysdc_update_enter;
360     uint64_t sysdc_update_exited;
361     uint64_t sysdc_update_not_sdc;
362     uint64_t sysdc_update_idle;
363     uint64_t sysdc_update_take_break;
364     uint64_t sysdc_update_no_psets;
365     uint64_t sysdc_tick_not_sdc;
366     uint64_t sysdc_tick_quantum_expired;
367     uint64_t sysdc_thread_enter_enter;
368 } sysdc_stats;
369 #endif

370 #ifdef UNCHANGED_PORTION_OMITTED
371 #endif

1291 /* --- consolidation-private interfaces --- */
1292 void
1293 sysdc_thread_enter(kthread_t *t, uint_t dc, uint_t flags)
1294 {
1295     void *buf = NULL;
1296     sysdc_params_t sdp;

1298     SYSDC_INC_STAT(sysdc_thread_enter_enter);

1300     ASSERT(sysdc_param_init);
1301     ASSERT(sysdccid >= 0);

1303     ASSERT((flags & ~SYSDC_THREAD_BATCH) == 0);

1305     sdp.sdp_minpri = sysdc_minpri;
1306     sdp.sdp_maxpri = sysdc_maxpri;
1307     sdp.sdp_DC = MAX(MIN(dc, sysdc_maxDC), sysdc_minDC);

1309     VERIFY0(CL_ALLOC(&buf, sysdccid, KM_SLEEP));
1308     VERIFY3U(CL_ALLOC(&buf, sysdccid, KM_SLEEP), ==, 0);

```

```
1311     ASSERT(t->t_lwp != NULL);
1312     ASSERT(t->t_cid == syscid);
1313     ASSERT(t->t_cldata == NULL);
1314     VERIFY0(CL_CANEXIT(t, NULL));
1315     VERIFY0(CL_ENTERCLASS(t, sysdclid, &sdp, kcred, buf));
1313     VERIFY3U(CL_CANEXIT(t, NULL), ==, 0);
1314     VERIFY3U(CL_ENTERCLASS(t, sysdclid, &sdp, kcred, buf), ==, 0);
1316     CL_EXITCLASS(syscid, NULL);
1317 }
```

\_\_\_\_\_unchanged\_portion\_omitted\_\_\_\_\_

```

*****
131604 Thu Jul 26 14:19:03 2012
new/usr/src/uts/common/fs/zfs/arc.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

997 static void
998 add_reference(arc_buf_hdr_t *ab, kmutex_t *hash_lock, void *tag)
999 {
1000     ASSERT(MUTEX_HELD(hash_lock));

1002     if ((refcount_add(&ab->b_refcnt, tag) == 1) &&
1003         (ab->b_state != arc_anon)) {
1004         uint64_t delta = ab->b_size * ab->b_datacnt;
1005         list_t *list = &ab->b_state->arcs_list[ab->b_type];
1006         uint64_t *size = &ab->b_state->arcs_lsize[ab->b_type];

1008         ASSERT(!MUTEX_HELD(&ab->b_state->arcs_mtx));
1009         mutex_enter(&ab->b_state->arcs_mtx);
1010         ASSERT(list_link_active(&ab->b_arc_node));
1011         list_remove(list, ab);
1012         if (GHOST_STATE(ab->b_state)) {
1013             ASSERT0(ab->b_datacnt);
1014             ASSERT3U(ab->b_datacnt, ==, 0);
1015             ASSERT3P(ab->b_buf, ==, NULL);
1016             delta = ab->b_size;
1017         }
1018         ASSERT(delta > 0);
1019         ASSERT3U(*size, >=, delta);
1020         atomic_add_64(size, -delta);
1021         mutex_exit(&ab->b_state->arcs_mtx);
1022         /* remove the prefetch flag if we get a reference */
1023         if (ab->b_flags & ARC_PREFETCH)
1024             ab->b_flags &= ~ARC_PREFETCH;
1025     }

_____unchanged_portion_omitted_____

1590 /*
1591 * Evict buffers from list until we've removed the specified number of
1592 * bytes. Move the removed buffers to the appropriate evict state.
1593 * If the recycle flag is set, then attempt to "recycle" a buffer:
1594 * - look for a buffer to evict that is 'bytes' long.
1595 * - return the data block from this buffer rather than freeing it.
1596 * This flag is used by callers that are trying to make space for a
1597 * new buffer in a full arc cache.
1598 *
1599 * This function makes a "best effort". It skips over any buffers
1600 * it can't get a hash_lock on, and so may not catch all candidates.
1601 * It may also return without evicting as much space as requested.
1602 */
1603 static void *
1604 arc_evict(arc_state_t *state, uint64_t spa, int64_t bytes, boolean_t recycle,
1605          arc_buf_contents_t type)
1606 {
1607     arc_state_t *evicted_state;
1608     uint64_t bytes_evicted = 0, skipped = 0, missed = 0;
1609     arc_buf_hdr_t *ab, *ab_prev = NULL;
1610     list_t *list = &state->arcs_list[type];
1611     kmutex_t *hash_lock;
1612     boolean_t have_lock;
1613     void *stolen = NULL;

1615     ASSERT(state == arc_mru || state == arc_mfu);

```

```

1617     evicted_state = (state == arc_mru) ? arc_mru_ghost : arc_mfu_ghost;

1619     mutex_enter(&state->arcs_mtx);
1620     mutex_enter(&evicted_state->arcs_mtx);

1622     for (ab = list_tail(list); ab; ab = ab_prev) {
1623         ab_prev = list_prev(list, ab);
1624         /* prefetch buffers have a minimum lifespan */
1625         if (HDR_IO_IN_PROGRESS(ab) ||
1626             (spa && ab->b_spa != spa) ||
1627             (ab->b_flags & (ARC_PREFETCH|ARC_INDIRECT) &&
1628              ddi_get_lbolt() - ab->b_arc_access <
1629              arc_min_prefetch_lifespan)) {
1630             skipped++;
1631             continue;
1632         }
1633         /* "lookahead" for better eviction candidate */
1634         if (recycle && ab->b_size != bytes &&
1635             ab_prev && ab_prev->b_size == bytes)
1636             continue;
1637         hash_lock = HDR_LOCK(ab);
1638         have_lock = MUTEX_HELD(hash_lock);
1639         if (have_lock || mutex_tryenter(hash_lock)) {
1640             ASSERT0(refcount_count(&ab->b_refcnt));
1641             ASSERT3U(refcount_count(&ab->b_refcnt), ==, 0);
1642             ASSERT(ab->b_datacnt > 0);
1643             while (ab->b_buf) {
1644                 arc_buf_t *buf = ab->b_buf;
1645                 if (!mutex_tryenter(&buf->b_evict_lock)) {
1646                     missed += 1;
1647                     break;
1648                 }
1649                 if (buf->b_data) {
1650                     bytes_evicted += ab->b_size;
1651                     if (recycle && ab->b_type == type &&
1652                         ab->b_size == bytes &&
1653                         !HDR_L2_WRITING(ab)) {
1654                         stolen = buf->b_data;
1655                         recycle = FALSE;
1656                     }
1657                 }
1658                 if (buf->b_efunc) {
1659                     mutex_enter(&arc_eviction_mtx);
1660                     arc_buf_destroy(buf,
1661                                     buf->b_data == stolen, FALSE);
1662                     ab->b_buf = buf->b_next;
1663                     buf->b_hdr = &arc_eviction_hdr;
1664                     buf->b_next = arc_eviction_list;
1665                     arc_eviction_list = buf;
1666                     mutex_exit(&arc_eviction_mtx);
1667                     mutex_exit(&buf->b_evict_lock);
1668                 } else {
1669                     mutex_exit(&buf->b_evict_lock);
1670                     arc_buf_destroy(buf,
1671                                     buf->b_data == stolen, TRUE);
1672                 }
1673             }

1674             if (ab->b_l2hdr) {
1675                 ARCSTAT_INCR(arcstat_evict_l2_cached,
1676                             ab->b_size);
1677             } else {
1678                 if (l2arc_write_eligible(ab->b_spa, ab)) {
1679                     ARCSTAT_INCR(arcstat_evict_l2_eligible,
1680                                 ab->b_size);
1681                 } else {

```

```

1682         ARCSTAT_INCR(
1683             arcstat_evict_l2_ineligible,
1684             ab->b_size);
1685     }
1686 }
1687
1688     if (ab->b_datacnt == 0) {
1689         arc_change_state(evicted_state, ab, hash_lock);
1690         ASSERT(HDR_IN_HASH_TABLE(ab));
1691         ab->b_flags |= ARC_IN_HASH_TABLE;
1692         ab->b_flags &= ~ARC_BUF_AVAILABLE;
1693         DTRACE_PROBE1(arc_evict, arc_buf_hdr_t *, ab);
1694     }
1695     if (!have_lock)
1696         mutex_exit(hash_lock);
1697     if (bytes >= 0 && bytes_evicted >= bytes)
1698         break;
1699 } else {
1700     missed += 1;
1701 }
1702 }
1703
1704 mutex_exit(&evicted_state->arcs_mtx);
1705 mutex_exit(&state->arcs_mtx);
1706
1707 if (bytes_evicted < bytes)
1708     dprintf("only evicted %lld bytes from %x",
1709         (longlong_t)bytes_evicted, state);
1710
1711 if (skipped)
1712     ARCSTAT_INCR(arcstat_evict_skip, skipped);
1713
1714 if (missed)
1715     ARCSTAT_INCR(arcstat_mutex_miss, missed);
1716
1717 /*
1718  * We have just evicted some data into the ghost state, make
1719  * sure we also adjust the ghost state size if necessary.
1720  */
1721 if (arc_no_grow &&
1722     arc_mru_ghost->arcs_size + arc_mfu_ghost->arcs_size > arc_c) {
1723     int64_t mru_over = arc_anon->arcs_size + arc_mru->arcs_size +
1724         arc_mru_ghost->arcs_size - arc_c;
1725
1726     if (mru_over > 0 && arc_mru_ghost->arcs_lsize[type] > 0) {
1727         int64_t todelete =
1728             MIN(arc_mru_ghost->arcs_lsize[type], mru_over);
1729         arc_evict_ghost(arc_mru_ghost, NULL, todelete);
1730     } else if (arc_mfu_ghost->arcs_lsize[type] > 0) {
1731         int64_t todelete = MIN(arc_mfu_ghost->arcs_lsize[type],
1732             arc_mru_ghost->arcs_size +
1733             arc_mfu_ghost->arcs_size - arc_c);
1734         arc_evict_ghost(arc_mfu_ghost, NULL, todelete);
1735     }
1736 }
1737
1738     return (stolen);
1739 }
1740
1741 _____unchanged_portion_omitted_____
1742
1743 2356 /*
1744 2357  * This routine is called whenever a buffer is accessed.
1745 2358  * NOTE: the hash lock is dropped in this function.
1746 2359  */
1747 2360 static void
1748 2361 arc_access(arc_buf_hdr_t *buf, kmutex_t *hash_lock)

```

```

2362 {
2363     clock_t now;
2364
2365     ASSERT(MUTEX_HELD(hash_lock));
2366
2367     if (buf->b_state == arc_anon) {
2368         /*
2369          * This buffer is not in the cache, and does not
2370          * appear in our "ghost" list. Add the new buffer
2371          * to the MRU state.
2372          */
2373
2374         ASSERT(buf->b_arc_access == 0);
2375         buf->b_arc_access = ddi_get_lbolt();
2376         DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
2377         arc_change_state(arc_mru, buf, hash_lock);
2378
2379     } else if (buf->b_state == arc_mru) {
2380         now = ddi_get_lbolt();
2381
2382         /*
2383          * If this buffer is here because of a prefetch, then either:
2384          * - clear the flag if this is a "referencing" read
2385          *   (any subsequent access will bump this into the MFU state).
2386          * or
2387          * - move the buffer to the head of the list if this is
2388          *   another prefetch (to make it less likely to be evicted).
2389          */
2390         if ((buf->b_flags & ARC_PREFETCH) != 0) {
2391             if (refcount_count(&buf->b_refcnt) == 0) {
2392                 ASSERT(list_link_active(&buf->b_arc_node));
2393             } else {
2394                 buf->b_flags &= ~ARC_PREFETCH;
2395                 ARCSTAT_BUMP(arcstat_mru_hits);
2396             }
2397             buf->b_arc_access = now;
2398             return;
2399         }
2400
2401         /*
2402          * This buffer has been "accessed" only once so far,
2403          * but it is still in the cache. Move it to the MFU
2404          * state.
2405          */
2406         if (now > buf->b_arc_access + ARC_MINTIME) {
2407             /*
2408              * More than 125ms have passed since we
2409              * instantiated this buffer. Move it to the
2410              * most frequently used state.
2411              */
2412             buf->b_arc_access = now;
2413             DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2414             arc_change_state(arc_mfu, buf, hash_lock);
2415         }
2416         ARCSTAT_BUMP(arcstat_mru_hits);
2417     } else if (buf->b_state == arc_mru_ghost) {
2418         arc_state_t *new_state;
2419         /*
2420          * This buffer has been "accessed" recently, but
2421          * was evicted from the cache. Move it to the
2422          * MFU state.
2423          */
2424
2425         if (buf->b_flags & ARC_PREFETCH) {
2426             new_state = arc_mru;
2427             if (refcount_count(&buf->b_refcnt) > 0)

```

```

2428     buf->b_flags &= ~ARC_PREFETCH;
2429     DTRACE_PROBE1(new_state__mru, arc_buf_hdr_t *, buf);
2430 } else {
2431     new_state = arc_mfu;
2432     DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2433 }
2434
2435     buf->b_arc_access = ddi_get_lbolt();
2436     arc_change_state(new_state, buf, hash_lock);
2437
2438     ARCSTAT_BUMP(arcstat_mru_ghost_hits);
2439 } else if (buf->b_state == arc_mfu) {
2440     /*
2441      * This buffer has been accessed more than once and is
2442      * still in the cache. Keep it in the MFU state.
2443      *
2444      * NOTE: an add_reference() that occurred when we did
2445      * the arc_read() will have kicked this off the list.
2446      * If it was a prefetch, we will explicitly move it to
2447      * the head of the list now.
2448      */
2449     if ((buf->b_flags & ARC_PREFETCH) != 0) {
2450         ASSERT(refcount_count(&buf->b_refcnt) == 0);
2451         ASSERT(list_link_active(&buf->b_arc_node));
2452     }
2453     ARCSTAT_BUMP(arcstat_mfu_hits);
2454     buf->b_arc_access = ddi_get_lbolt();
2455 } else if (buf->b_state == arc_mfu_ghost) {
2456     arc_state_t *new_state = arc_mfu;
2457     /*
2458      * This buffer has been accessed more than once but has
2459      * been evicted from the cache. Move it back to the
2460      * MFU state.
2461      */
2462     if (buf->b_flags & ARC_PREFETCH) {
2463         /*
2464          * This is a prefetch access...
2465          * move this block back to the MRU state.
2466          */
2467         ASSERT0(refcount_count(&buf->b_refcnt));
2468         ASSERT3U(refcount_count(&buf->b_refcnt), ==, 0);
2469         new_state = arc_mru;
2470     }
2471
2472     buf->b_arc_access = ddi_get_lbolt();
2473     DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2474     arc_change_state(new_state, buf, hash_lock);
2475
2476     ARCSTAT_BUMP(arcstat_mfu_ghost_hits);
2477 } else if (buf->b_state == arc_l2c_only) {
2478     /*
2479      * This buffer is on the 2nd Level ARC.
2480      */
2481
2482     buf->b_arc_access = ddi_get_lbolt();
2483     DTRACE_PROBE1(new_state__mfu, arc_buf_hdr_t *, buf);
2484     arc_change_state(arc_mfu, buf, hash_lock);
2485 } else {
2486     ASSERT(!"invalid arc state");
2487 }
2488 }

```

\_\_\_\_\_unchanged portion omitted\_\_\_\_\_

```

2687 int
2688 arc_read_nolock(zio_t *pio, spa_t *spa, const blkptr_t *bp,

```

```

2689     arc_done_func_t *done, void *private, int priority, int zio_flags,
2690     uint32_t *arc_flags, const zbookmark_t *zb)
2691 {
2692     arc_buf_hdr_t *hdr;
2693     arc_buf_t *buf;
2694     kmutex_t *hash_lock;
2695     zio_t *rzio;
2696     uint64_t guid = spa_load_guid(spa);
2697
2698 top:
2699     hdr = buf_hash_find(guid, BP_IDENTITY(bp), BP_PHYSICAL_BIRTH(bp),
2700     &hash_lock);
2701     if (hdr && hdr->b_datacnt > 0) {
2702
2703         *arc_flags |= ARC_CACHED;
2704
2705         if (HDR_IO_IN_PROGRESS(hdr)) {
2706             if (*arc_flags & ARC_WAIT) {
2707                 cv_wait(&hdr->b_cv, hash_lock);
2708                 mutex_exit(hash_lock);
2709                 goto top;
2710             }
2711             ASSERT(*arc_flags & ARC_NOWAIT);
2712
2713             if (done) {
2714                 arc_callback_t *acb = NULL;
2715
2716                 acb = kmem_zalloc(sizeof(arc_callback_t),
2717                 KM_SLEEP);
2718                 acb->acb_done = done;
2719                 acb->acb_private = private;
2720                 if (pio != NULL)
2721                     acb->acb_zio_dummy = zio_null(pio,
2722                     spa, NULL, NULL, NULL, zio_flags);
2723
2724                 ASSERT(acb->acb_done != NULL);
2725                 acb->acb_next = hdr->b_acb;
2726                 hdr->b_acb = acb;
2727                 add_reference(hdr, hash_lock, private);
2728                 mutex_exit(hash_lock);
2729                 return (0);
2730             }
2731             mutex_exit(hash_lock);
2732             return (0);
2733         }
2734     }
2735
2736     ASSERT(hdr->b_state == arc_mru || hdr->b_state == arc_mfu);
2737
2738     if (done) {
2739         add_reference(hdr, hash_lock, private);
2740         /*
2741          * If this block is already in use, create a new
2742          * copy of the data so that we will be guaranteed
2743          * that arc_release() will always succeed.
2744          */
2745         buf = hdr->b_buf;
2746         ASSERT(buf);
2747         ASSERT(buf->b_data);
2748         if (HDR_BUF_AVAILABLE(hdr)) {
2749             ASSERT(buf->b_efunc == NULL);
2750             hdr->b_flags &= ~ARC_BUF_AVAILABLE;
2751         } else {
2752             buf = arc_buf_clone(buf);
2753         }
2754     }

```

```

2755     } else if (*arc_flags & ARC_PREFETCH &&
2756               refcount_count(&hdr->b_refcnt) == 0) {
2757         hdr->b_flags |= ARC_PREFETCH;
2758     }
2759     DTRACE_PROBE1(arc_hit, arc_buf_hdr_t *, hdr);
2760     arc_access(hdr, hash_lock);
2761     if (*arc_flags & ARC_L2CACHE)
2762         hdr->b_flags |= ARC_L2CACHE;
2763     mutex_exit(hash_lock);
2764     ARCSTAT_BUMP(arcstat_hits);
2765     ARCSTAT_CONDDSTAT(!(hdr->b_flags & ARC_PREFETCH),
2766                       demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
2767                       data, metadata, hits);

2769     if (done)
2770         done(NULL, buf, private);
2771 } else {
2772     uint64_t size = BP_GET_LSIZE(bp);
2773     arc_callback_t *acb;
2774     vdev_t *vd = NULL;
2775     uint64_t addr;
2776     boolean_t devw = B_FALSE;

2778     if (hdr == NULL) {
2779         /* this block is not in the cache */
2780         arc_buf_hdr_t *exists;
2781         arc_buf_contents_t type = BP_GET_BUFC_TYPE(bp);
2782         buf = arc_buf_alloc(spa, size, private, type);
2783         hdr = buf->b_hdr;
2784         hdr->b_dva = *BP_IDENTITY(bp);
2785         hdr->b_birth = BP_PHYSICAL_BIRTH(bp);
2786         hdr->b_cksum0 = bp->blk_cksum.zc_word[0];
2787         exists = buf_hash_insert(hdr, &hash_lock);
2788         if (exists) {
2789             /* somebody beat us to the hash insert */
2790             mutex_exit(hash_lock);
2791             buf_discard_identity(hdr);
2792             (void) arc_buf_remove_ref(buf, private);
2793             goto top; /* restart the IO request */
2794         }
2795         /* if this is a prefetch, we don't have a reference */
2796         if (*arc_flags & ARC_PREFETCH) {
2797             (void) remove_reference(hdr, hash_lock,
2798                                   private);
2799             hdr->b_flags |= ARC_PREFETCH;
2800         }
2801         if (*arc_flags & ARC_L2CACHE)
2802             hdr->b_flags |= ARC_L2CACHE;
2803         if (BP_GET_LEVEL(bp) > 0)
2804             hdr->b_flags |= ARC_INDIRECT;
2805     } else {
2806         /* this block is in the ghost cache */
2807         ASSERT(GHOST_STATE(hdr->b_state));
2808         ASSERT(!HDR_IO_IN_PROGRESS(hdr));
2809         ASSERT0(refcount_count(&hdr->b_refcnt));
2810         ASSERT3U(refcount_count(&hdr->b_refcnt), ==, 0);
2811         ASSERT(hdr->b_buf == NULL);

2812         /* if this is a prefetch, we don't have a reference */
2813         if (*arc_flags & ARC_PREFETCH)
2814             hdr->b_flags |= ARC_PREFETCH;
2815         else
2816             add_reference(hdr, hash_lock, private);
2817         if (*arc_flags & ARC_L2CACHE)
2818             hdr->b_flags |= ARC_L2CACHE;
2819         buf = kmem_cache_alloc(buf_cache, KM_PUSHPAGE);

```

```

2820         buf->b_hdr = hdr;
2821         buf->b_data = NULL;
2822         buf->b_efunc = NULL;
2823         buf->b_private = NULL;
2824         buf->b_next = NULL;
2825         hdr->b_buf = buf;
2826         ASSERT(hdr->b_datacnt == 0);
2827         hdr->b_datacnt = 1;
2828         arc_get_data_buf(buf);
2829         arc_access(hdr, hash_lock);
2830     }

2832     ASSERT(!GHOST_STATE(hdr->b_state));

2834     acb = kmem_zalloc(sizeof(arc_callback_t), KM_SLEEP);
2835     acb->acb_done = done;
2836     acb->acb_private = private;

2838     ASSERT(hdr->b_acb == NULL);
2839     hdr->b_acb = acb;
2840     hdr->b_flags |= ARC_IO_IN_PROGRESS;

2842     if (HDR_L2CACHE(hdr) && hdr->b_l2hdr != NULL &&
2843         (vd = hdr->b_l2hdr->b_dev->l2ad_vdev) != NULL) {
2844         devw = hdr->b_l2hdr->b_dev->l2ad_writing;
2845         addr = hdr->b_l2hdr->b_daddr;
2846         /*
2847          * Lock out device removal.
2848          */
2849         if (vdev_is_dead(vd) ||
2850             !spa_config_tryenter(spa, SCL_L2ARC, vd, RW_READER))
2851             vd = NULL;
2852     }

2854     mutex_exit(hash_lock);

2856     ASSERT3U(hdr->b_size, ==, size);
2857     DTRACE_PROBE4(arc_miss, arc_buf_hdr_t *, hdr, blkptr_t *, bp,
2858                 uint64_t, size, zbookmark_t *, zb);
2859     ARCSTAT_BUMP(arcstat_misses);
2860     ARCSTAT_CONDDSTAT(!(hdr->b_flags & ARC_PREFETCH),
2861                       demand, prefetch, hdr->b_type != ARC_BUFC_METADATA,
2862                       data, metadata, misses);

2864     if (vd != NULL && l2arc_ndev != 0 && !(l2arc_norw && devw)) {
2865         /*
2866          * Read from the L2ARC if the following are true:
2867          * 1. The L2ARC vdev was previously cached.
2868          * 2. This buffer still has L2ARC metadata.
2869          * 3. This buffer isn't currently writing to the L2ARC.
2870          * 4. The L2ARC entry wasn't evicted, which may
2871          *    also have invalidated the vdev.
2872          * 5. This isn't prefetch and l2arc_noprefetch is set.
2873          */
2874         if (hdr->b_l2hdr != NULL &&
2875             !HDR_L2_WRITING(hdr) && !HDR_L2_EVICTED(hdr) &&
2876             !(l2arc_noprefetch && HDR_PREFETCH(hdr))) {
2877             l2arc_read_callback_t *cb;

2879             DTRACE_PROBE1(l2arc_hit, arc_buf_hdr_t *, hdr);
2880             ARCSTAT_BUMP(arcstat_l2_hits);

2882             cb = kmem_zalloc(sizeof(l2arc_read_callback_t),
2883                             KM_SLEEP);
2884             cb->l2rcb_buf = buf;
2885             cb->l2rcb_spa = spa;

```

```

2886     cb->l2rcb_bp = *bp;
2887     cb->l2rcb_zb = *zb;
2888     cb->l2rcb_flags = zio_flags;

2890     /*
2891     * l2arc read. The SCL_L2ARC lock will be
2892     * released by l2arc_read_done().
2893     */
2894     rzio = zio_read_phys(pio, vd, addr, size,
2895         buf->b_data, ZIO_CHECKSUM_OFF,
2896         l2arc_read_done, cb, priority, zio_flags |
2897         ZIO_FLAG_DONT_CACHE | ZIO_FLAG_CANFAIL |
2898         ZIO_FLAG_DONT_PROPAGATE |
2899         ZIO_FLAG_DONT_RETRY, B_FALSE);
2900     DTRACE_PROBE2(l2arc_read, vdev_t *, vd,
2901         zio_t *, rzio);
2902     ARCSTAT_INCR(arcstat_l2_read_bytes, size);

2904     if (*arc_flags & ARC_NOWAIT) {
2905         zio_nowait(rzio);
2906         return (0);
2907     }

2909     ASSERT(*arc_flags & ARC_WAIT);
2910     if (zio_wait(rzio) == 0)
2911         return (0);

2913     /* l2arc read error; goto zio_read() */
2914     } else {
2915         DTRACE_PROBE1(l2arc__miss,
2916             arc_buf_hdr_t *, hdr);
2917         ARCSTAT_BUMP(arcstat_l2_misses);
2918         if (HDR_L2_WRITING(hdr))
2919             ARCSTAT_BUMP(arcstat_l2_rw_clash);
2920         spa_config_exit(spa, SCL_L2ARC, vd);
2921     }
2922     } else {
2923         if (vd != NULL)
2924             spa_config_exit(spa, SCL_L2ARC, vd);
2925         if (l2arc_ndev != 0) {
2926             DTRACE_PROBE1(l2arc__miss,
2927                 arc_buf_hdr_t *, hdr);
2928             ARCSTAT_BUMP(arcstat_l2_misses);
2929         }
2930     }

2932     rzio = zio_read(pio, spa, bp, buf->b_data, size,
2933         arc_read_done, buf, priority, zio_flags, zb);

2935     if (*arc_flags & ARC_WAIT)
2936         return (zio_wait(rzio));

2938     ASSERT(*arc_flags & ARC_NOWAIT);
2939     zio_nowait(rzio);
2940     }
2941     return (0);
2942 }

```

unchanged portion omitted

```

4236 /*
4237 * Find and write ARC buffers to the L2ARC device.
4238 *
4239 * An ARC_L2_WRITING flag is set so that the L2ARC buffers are not valid
4240 * for reading until they have completed writing.
4241 */
4242 static uint64_t

```

```

4243 l2arc_write_buffers(spa_t *spa, l2arc_dev_t *dev, uint64_t target_sz)
4244 {
4245     arc_buf_hdr_t *ab, *ab_prev, *head;
4246     l2arc_buf_hdr_t *hdr_l2;
4247     list_t *list;
4248     uint64_t passed_sz, write_sz, buf_sz, headroom;
4249     void *buf_data;
4250     kmutex_t *hash_lock, *list_lock;
4251     boolean_t have_lock, full;
4252     l2arc_write_callback_t *cb;
4253     zio_t *pio, *wzio;
4254     uint64_t guid = spa_load_guid(spa);

4256     ASSERT(dev->l2ad_vdev != NULL);

4258     pio = NULL;
4259     write_sz = 0;
4260     full = B_FALSE;
4261     head = kmem_cache_alloc(hdr_cache, KM_PUSHPAGE);
4262     head->b_flags |= ARC_L2_WRITE_HEAD;

4264     /*
4265     * Copy buffers for L2ARC writing.
4266     */
4267     mutex_enter(&l2arc_buflist_mtx);
4268     for (int try = 0; try <= 3; try++) {
4269         list = l2arc_list_locked(try, &list_lock);
4270         passed_sz = 0;

4272         /*
4273         * L2ARC fast warmup.
4274         *
4275         * Until the ARC is warm and starts to evict, read from the
4276         * head of the ARC lists rather than the tail.
4277         */
4278         headroom = target_sz * l2arc_headroom;
4279         if (arc_warm == B_FALSE)
4280             ab = list_head(list);
4281         else
4282             ab = list_tail(list);

4284         for (; ab; ab = ab_prev) {
4285             if (arc_warm == B_FALSE)
4286                 ab_prev = list_next(list, ab);
4287             else
4288                 ab_prev = list_prev(list, ab);

4290             hash_lock = HDR_LOCK(ab);
4291             have_lock = MUTEX_HELD(hash_lock);
4292             if (!have_lock && !mutex_tryenter(hash_lock)) {
4293                 /*
4294                 * Skip this buffer rather than waiting.
4295                 */
4296                 continue;
4297             }

4299             passed_sz += ab->b_size;
4300             if (passed_sz > headroom) {
4301                 /*
4302                 * Searched too far.
4303                 */
4304                 mutex_exit(hash_lock);
4305                 break;
4306             }
4308             if (!l2arc_write_eligible(guid, ab)) {

```



```

4309         mutex_exit(hash_lock);
4310         continue;
4311     }
4312
4313     if ((write_sz + ab->b_size) > target_sz) {
4314         full = B_TRUE;
4315         mutex_exit(hash_lock);
4316         break;
4317     }
4318
4319     if (pio == NULL) {
4320         /*
4321          * Insert a dummy header on the buflist so
4322          * l2arc_write_done() can find where the
4323          * write buffers begin without searching.
4324          */
4325         list_insert_head(dev->l2ad_buflist, head);
4326
4327         cb = kmem_alloc(
4328             sizeof (l2arc_write_callback_t), KM_SLEEP);
4329         cb->l2wcb_dev = dev;
4330         cb->l2wcb_head = head;
4331         pio = zio_root(spa, l2arc_write_done, cb,
4332             ZIO_FLAG_CANFAIL);
4333     }
4334
4335     /*
4336      * Create and add a new L2ARC header.
4337      */
4338     hdr12 = kmem_zalloc(sizeof (l2arc_buf_hdr_t), KM_SLEEP);
4339     hdr12->b_dev = dev;
4340     hdr12->b_daddr = dev->l2ad_hand;
4341
4342     ab->b_flags |= ARC_L2_WRITING;
4343     ab->b_l2hdr = hdr12;
4344     list_insert_head(dev->l2ad_buflist, ab);
4345     buf_data = ab->b_buf->b_data;
4346     buf_sz = ab->b_size;
4347
4348     /*
4349      * Compute and store the buffer cksum before
4350      * writing. On debug the cksum is verified first.
4351      */
4352     arc_cksum_verify(ab->b_buf);
4353     arc_cksum_compute(ab->b_buf, B_TRUE);
4354
4355     mutex_exit(hash_lock);
4356
4357     wzio = zio_write_phys(pio, dev->l2ad_vdev,
4358         dev->l2ad_hand, buf_sz, buf_data, ZIO_CHECKSUM_OFF,
4359         NULL, NULL, ZIO_PRIORITY_ASYNC_WRITE,
4360         ZIO_FLAG_CANFAIL, B_FALSE);
4361
4362     DTRACE_PROBE2(l2arc_write, vdev_t *, dev->l2ad_vdev,
4363         zio_t *, wzio);
4364     (void) zio_nowait(wzio);
4365
4366     /*
4367      * Keep the clock hand suitably device-aligned.
4368      */
4369     buf_sz = vdev_psize_to_asize(dev->l2ad_vdev, buf_sz);
4370
4371     write_sz += buf_sz;
4372     dev->l2ad_hand += buf_sz;
4373 }

```

```

4375         mutex_exit(list_lock);
4376
4377         if (full == B_TRUE)
4378             break;
4379     }
4380     mutex_exit(&l2arc_buflist_mtx);
4381
4382     if (pio == NULL) {
4383         ASSERT0(write_sz);
4384         ASSERT3U(write_sz, ==, 0);
4385         kmem_cache_free(hdr_cache, head);
4386         return (0);
4387     }
4388     ASSERT3U(write_sz, <=, target_sz);
4389     ARCSTAT_BUMP(arcstat_l2_writes_sent);
4390     ARCSTAT_INCR(arcstat_l2_write_bytes, write_sz);
4391     ARCSTAT_INCR(arcstat_l2_size, write_sz);
4392     vdev_space_update(dev->l2ad_vdev, write_sz, 0, 0);
4393
4394     /*
4395      * Bump device hand to the device start if it is approaching the end.
4396      * l2arc_evict() will already have evicted ahead for this case.
4397      */
4398     if (dev->l2ad_hand >= (dev->l2ad_end - target_sz)) {
4399         vdev_space_update(dev->l2ad_vdev,
4400             dev->l2ad_end - dev->l2ad_hand, 0, 0);
4401         dev->l2ad_hand = dev->l2ad_start;
4402         dev->l2ad_evict = dev->l2ad_start;
4403         dev->l2ad_first = B_FALSE;
4404     }
4405
4406     dev->l2ad_writing = B_TRUE;
4407     (void) zio_wait(pio);
4408     dev->l2ad_writing = B_FALSE;
4409
4410     return (write_sz);
4411 }
_____unchanged_portion_omitted_____

```

```

*****
13003 Thu Jul 26 14:19:04 2012
new/usr/src/uts/common/fs/zfs/bpobj.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 * Copyright (c) 2011 by Delphix. All rights reserved.
25 */
26 #include <sys/bpobj.h>
27 #include <sys/zfs_context.h>
28 #include <sys/refcount.h>
29 #include <sys/dsl_pool.h>
30
31 uint64_t
32 bpobj_alloc(objset_t *os, int blocksize, dmu_tx_t *tx)
33 {
34     int size;
35
36     if (spa_version(dmu_objset_spa(os)) < SPA_VERSION_BPOBJ_ACCOUNT)
37         size = BPOBJ_SIZE_V0;
38     else if (spa_version(dmu_objset_spa(os)) < SPA_VERSION_DEADLISTS)
39         size = BPOBJ_SIZE_V1;
40     else
41         size = sizeof (bpobj_phys_t);
42
43     return (dmu_object_alloc(os, DMU_OT_BPOBJ, blocksize,
44         DMU_OT_BPOBJ_HDR, size, tx));
45 }
46
47 void
48 bpobj_free(objset_t *os, uint64_t obj, dmu_tx_t *tx)
49 {
50     int64_t i;
51     bpobj_t bpo;
52     dmu_object_info_t doi;
53     int epb;
54     dmu_buf_t *dbuf = NULL;
55
56     VERIFY0(bpobj_open(&bpo, os, obj));
57     VERIFY3U(0, ==, bpobj_open(&bpo, os, obj));
58
59     mutex_enter(&bpo.bpo_lock);

```

```

60     if (!bpo.bpo_havesubobj || bpo.bpo_phys->bpo_subobjs == 0)
61         goto out;
62
63     VERIFY0(dmu_object_info(os, bpo.bpo_phys->bpo_subobjs, &doi));
64     VERIFY3U(0, ==, dmu_object_info(os, bpo.bpo_phys->bpo_subobjs, &doi));
65     epb = doi.doi_data_block_size / sizeof (uint64_t);
66
67     for (i = bpo.bpo_phys->bpo_num_subobjs - 1; i >= 0; i--) {
68         uint64_t *objarray;
69         uint64_t offset, blkoff;
70
71         offset = i * sizeof (uint64_t);
72         blkoff = P2PHASE(i, epb);
73
74         if (dbuf == NULL || dbuf->db_offset > offset) {
75             if (dbuf)
76                 dmu_buf_rele(dbuf, FTAG);
77             VERIFY0(dmu_buf_hold(os,
78                 VERIFY3U(0, ==, dmu_buf_hold(os,
79                     bpo.bpo_phys->bpo_subobjs, offset, FTAG, &dbuf, 0)));
80
81             ASSERT3U(offset, >=, dbuf->db_offset);
82             ASSERT3U(offset, <, dbuf->db_offset + dbuf->db_size);
83
84             objarray = dbuf->db_data;
85             bpobj_free(os, objarray[blkoff], tx);
86         }
87         if (dbuf) {
88             dmu_buf_rele(dbuf, FTAG);
89             dbuf = NULL;
90         }
91     }
92     VERIFY0(dmu_object_free(os, bpo.bpo_phys->bpo_subobjs, tx));
93     VERIFY3U(0, ==, dmu_object_free(os, bpo.bpo_phys->bpo_subobjs, tx));
94
95 out:
96     mutex_exit(&bpo.bpo_lock);
97     bpobj_close(&bpo);
98
99     VERIFY0(dmu_object_free(os, obj, tx));
100    VERIFY3U(0, ==, dmu_object_free(os, obj, tx));
101
102    unchanged portion omitted
103
104    149 static int
105    150 bpobj_iterate_impl(bpobj_t *bpo, bpobj_iterator_t func, void *arg, dmu_tx_t *tx,
106    151     boolean_t free)
107    152 {
108    153     dmu_object_info_t doi;
109    154     int epb;
110    155     int64_t i;
111    156     int err = 0;
112    157     dmu_buf_t *dbuf = NULL;
113
114    159     mutex_enter(&bpo->bpo_lock);
115
116    161     if (free)
117    162         dmu_buf_will_dirty(bpo->bpo_dbuf, tx);
118
119    164     for (i = bpo->bpo_phys->bpo_num_blkptrs - 1; i >= 0; i--) {
120    165         blkptr_t *bparray;
121    166         blkptr_t *bp;
122    167         uint64_t offset, blkoff;
123
124    169         offset = i * sizeof (blkptr_t);
125    170         blkoff = P2PHASE(i, bpo->bpo_epb);

```

```

172         if (dbuf == NULL || dbuf->db_offset > offset) {
173             if (dbuf)
174                 dmu_buf_rele(dbuf, FTAG);
175             err = dmu_buf_hold(bpo->bpo_os, bpo->bpo_object, offset,
176                             FTAG, &dbuf, 0);
177             if (err)
178                 break;
179         }
181     ASSERT3U(offset, >=, dbuf->db_offset);
182     ASSERT3U(offset, <, dbuf->db_offset + dbuf->db_size);
184     bpararray = dbuf->db_data;
185     bp = &bpararray[blkoff];
186     err = func(arg, bp, tx);
187     if (err)
188         break;
189     if (free) {
190         bpo->bpo_phys->bpo_bytes -=
191             bp_get_dsize_sync(dmu_objset_spa(bpo->bpo_os), bp);
192         ASSERT3S(bpo->bpo_phys->bpo_bytes, >=, 0);
193         if (bpo->bpo_havecomp) {
194             bpo->bpo_phys->bpo_comp -= BP_GET_PSIZE(bp);
195             bpo->bpo_phys->bpo_uncomp -= BP_GET_UCSIZE(bp);
196         }
197         bpo->bpo_phys->bpo_num_blkptrs--;
198         ASSERT3S(bpo->bpo_phys->bpo_num_blkptrs, >=, 0);
199     }
200 }
201 if (dbuf) {
202     dmu_buf_rele(dbuf, FTAG);
203     dbuf = NULL;
204 }
205 if (free) {
206     i++;
207     VERIFY0(dmu_free_range(bpo->bpo_os, bpo->bpo_object,
208                          VERIFY3U(0, ==, dmu_free_range(bpo->bpo_os, bpo->bpo_object,
209                                                         i * sizeof(blkptr_t), -1ULL, tx)));
210 }
211 if (err || !bpo->bpo_havesubobj || bpo->bpo_phys->bpo_subobjs == 0)
212     goto out;
213
214 ASSERT(bpo->bpo_havecomp);
215 err = dmu_object_info(bpo->bpo_os, bpo->bpo_phys->bpo_subobjs, &doi);
216 if (err) {
217     mutex_exit(&bpo->bpo_lock);
218     return (err);
219 }
220 epb = doi.doi_data_block_size / sizeof(uint64_t);
221
222 for (i = bpo->bpo_phys->bpo_num_subobjs - 1; i >= 0; i--) {
223     uint64_t *objarray;
224     uint64_t offset, blkoff;
225     bpobj_t sublist;
226     uint64_t used_before, comp_before, uncomp_before;
227     uint64_t used_after, comp_after, uncomp_after;
228
229     offset = i * sizeof(uint64_t);
230     blkoff = P2PHASE(i, epb);
231
232     if (dbuf == NULL || dbuf->db_offset > offset) {
233         if (dbuf)
234             dmu_buf_rele(dbuf, FTAG);
235         err = dmu_buf_hold(bpo->bpo_os,
236                           bpo->bpo_phys->bpo_subobjs, offset, FTAG, &dbuf, 0);

```

```

236         if (err)
237             break;
238     }
240     ASSERT3U(offset, >=, dbuf->db_offset);
241     ASSERT3U(offset, <, dbuf->db_offset + dbuf->db_size);
243     objarray = dbuf->db_data;
244     err = bpobj_open(&sublist, bpo->bpo_os, objarray[blkoff]);
245     if (err)
246         break;
247     if (free) {
248         err = bpobj_space(&sublist,
249                          &used_before, &comp_before, &uncomp_before);
250         if (err)
251             break;
252     }
253     err = bpobj_iterate_impl(&sublist, func, arg, tx, free);
254     if (free) {
255         VERIFY0(bpobj_space(&sublist,
256                            VERIFY3U(0, ==, bpobj_space(&sublist,
257                                                         &used_after, &comp_after, &uncomp_after)));
258         bpo->bpo_phys->bpo_bytes -= used_before - used_after;
259         ASSERT3S(bpo->bpo_phys->bpo_bytes, >=, 0);
260         bpo->bpo_phys->bpo_comp -= comp_before - comp_after;
261         bpo->bpo_phys->bpo_uncomp -=
262             uncomp_before - uncomp_after;
263     }
264     bpobj_close(&sublist);
265     if (err)
266         break;
267     if (free) {
268         err = dmu_object_free(bpo->bpo_os,
269                               objarray[blkoff], tx);
270         if (err)
271             break;
272         bpo->bpo_phys->bpo_num_subobjs--;
273         ASSERT3S(bpo->bpo_phys->bpo_num_subobjs, >=, 0);
274     }
275 }
276 if (dbuf) {
277     dmu_buf_rele(dbuf, FTAG);
278     dbuf = NULL;
279 }
280 if (free) {
281     VERIFY0(dmu_free_range(bpo->bpo_os,
282                          VERIFY3U(0, ==, dmu_free_range(bpo->bpo_os,
283                                                         bpo->bpo_phys->bpo_subobjs,
284                                                         (i + 1) * sizeof(uint64_t), -1ULL, tx)));
285 }
286 out:
287 /* If there are no entries, there should be no bytes. */
288 ASSERT(bpo->bpo_phys->bpo_num_blkptrs > 0 ||
289        (bpo->bpo_havesubobj && bpo->bpo_phys->bpo_num_subobjs > 0) ||
290        bpo->bpo_phys->bpo_bytes == 0);
292     mutex_exit(&bpo->bpo_lock);
293     return (err);
294 }
295
296 unchanged_portion_omitted
297
298 void
299 bpobj_enqueue_subobj(bpobj_t *bpo, uint64_t subobj, dmu_tx_t *tx)
300 {

```

```

318     bpobj_t subbpo;
319     uint64_t used, comp, uncomp, subsubobj;

321     ASSERT(bpo->bpo_havesubobj);
322     ASSERT(bpo->bpo_havecomp);

324     VERIFY0(bpobj_open(&subbpo, bpo->bpo_os, subobj));
325     VERIFY0(bpobj_space(&subbpo, &used, &comp, &uncomp));
326     VERIFY3U(0, ==, bpobj_open(&subbpo, bpo->bpo_os, subobj));
327     VERIFY3U(0, ==, bpobj_space(&subbpo, &used, &comp, &uncomp));

327     if (used == 0) {
328         /* No point in having an empty subobj. */
329         bpobj_close(&subbpo);
330         bpobj_free(bpo->bpo_os, subobj, tx);
331         return;
332     }

334     dmu_buf_will_dirty(bpo->bpo_dbuf, tx);
335     if (bpo->bpo_phys->bpo_subobjs == 0) {
336         bpo->bpo_phys->bpo_subobjs = dmu_object_alloc(bpo->bpo_os,
337             DMU_OT_BPOBJ_SUBOBJ, SPA_MAXBLOCKSIZE, DMU_OT_NONE, 0, tx);
338     }

340     mutex_enter(&bpo->bpo_lock);
341     dmu_write(bpo->bpo_os, bpo->bpo_phys->bpo_subobjs,
342         bpo->bpo_phys->bpo_num_subobjs * sizeof(subobj),
343         sizeof(subobj), &subobj, tx);
344     bpo->bpo_phys->bpo_num_subobjs++;

346     /*
347      * If subobj has only one block of subobjs, then move subobj's
348      * subobjs to bpo's subobj list directly. This reduces
349      * recursion in bpobj_iterate due to nested subobjs.
350      */
351     subsubobjs = subbpo.bpo_phys->bpo_subobjs;
352     if (subsubobjs != 0) {
353         dmu_object_info_t doi;

355         VERIFY0(dmu_object_info(bpo->bpo_os, subsubobjs, &doi));
356         VERIFY3U(0, ==, dmu_object_info(bpo->bpo_os, subsubobjs, &doi));
357         if (doi.doi_max_offset == doi.doi_data_block_size) {
358             dmu_buf_t *subdb;
359             uint64_t numsubsub = subbpo.bpo_phys->bpo_num_subobjs;

360             VERIFY0(dmu_buf_hold(bpo->bpo_os, subsubobjs,
361                 VERIFY3U(0, ==, dmu_buf_hold(bpo->bpo_os, subsubobjs,
362                     0, FTAG, &subdb, 0)));
363             dmu_write(bpo->bpo_os, bpo->bpo_phys->bpo_subobjs,
364                 bpo->bpo_phys->bpo_num_subobjs * sizeof(subobj),
365                 numsubsub * sizeof(subobj), subdb->db_data, tx);
366             dmu_buf_rele(subdb, FTAG);
367             bpo->bpo_phys->bpo_num_subobjs += numsubsub;

368             dmu_buf_will_dirty(subbpo.bpo_dbuf, tx);
369             subbpo.bpo_phys->bpo_subobjs = 0;
370             VERIFY0(dmu_object_free(bpo->bpo_os,
371                 VERIFY3U(0, ==, dmu_object_free(bpo->bpo_os,
372                     subsubobjs, tx)));
373         }
374     }
375     bpo->bpo_phys->bpo_bytes += used;
376     bpo->bpo_phys->bpo_comp += comp;
377     bpo->bpo_phys->bpo_uncomp += uncomp;
378     mutex_exit(&bpo->bpo_lock);

```

```

379         bpobj_close(&subbpo);
380     }

382 void
383 bpobj_enqueue(bpobj_t *bpo, const blkptr_t *bp, dmu_tx_t *tx)
384 {
385     blkptr_t stored_bp = *bp;
386     uint64_t offset;
387     int blkoff;
388     blkptr_t *bparray;

390     ASSERT(!BP_IS_HOLE(bp));

392     /* We never need the fill count. */
393     stored_bp.blk_fill = 0;

395     /* The bpobj will compress better if we can leave off the checksum */
396     if (!BP_GET_DEDUP(bp))
397         bzero(&stored_bp.blk_cksum, sizeof(stored_bp.blk_cksum));

399     mutex_enter(&bpo->bpo_lock);

401     offset = bpo->bpo_phys->bpo_num_blkptrs * sizeof(stored_bp);
402     blkoff = P2PHASE(bpo->bpo_phys->bpo_num_blkptrs, bpo->bpo_epb);

404     if (bpo->bpo_cached_dbuf == NULL ||
405         offset < bpo->bpo_cached_dbuf->db_offset ||
406         offset >= bpo->bpo_cached_dbuf->db_offset +
407             bpo->bpo_cached_dbuf->db_size) {
408         if (bpo->bpo_cached_dbuf)
409             dmu_buf_rele(bpo->bpo_cached_dbuf, bpo);
410         VERIFY0(dmu_buf_hold(bpo->bpo_os, bpo->bpo_object,
411             VERIFY3U(0, ==, dmu_buf_hold(bpo->bpo_os, bpo->bpo_object,
412                 offset, bpo, &bpo->bpo_cached_dbuf, 0)));
413     }

414     dmu_buf_will_dirty(bpo->bpo_cached_dbuf, tx);
415     bparray = bpo->bpo_cached_dbuf->db_data;
416     bparray[blkoff] = stored_bp;

418     dmu_buf_will_dirty(bpo->bpo_dbuf, tx);
419     bpo->bpo_phys->bpo_num_blkptrs++;
420     bpo->bpo_phys->bpo_bytes +=
421         bp_get_dsize_sync(dmu_objset_spa(bpo->bpo_os), bp);
422     if (bpo->bpo_havecomp) {
423         bpo->bpo_phys->bpo_comp += BP_GET_PSIZE(bp);
424         bpo->bpo_phys->bpo_uncomp += BP_GET_UCSIZE(bp);
425     }
426     mutex_exit(&bpo->bpo_lock);
427 }

```

unchanged portion omitted

```

*****
5944 Thu Jul 26 14:19:05 2012
new/usr/src/uts/common/fs/zfs/bptree.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

60 uint64_t
61 bptree_alloc(objset_t *os, dmu_tx_t *tx)
62 {
63     uint64_t obj;
64     dmu_buf_t *db;
65     bptree_phys_t *bt;

67     obj = dmu_object_alloc(os, DMU_OTN_UINT64_METADATA,
68         SPA_MAXBLOCKSIZE, DMU_OTN_UINT64_METADATA,
69         sizeof (bptree_phys_t), tx);

71     /*
72      * Bonus buffer contents are already initialized to 0, but for
73      * readability we make it explicit.
74      */
75     VERIFY0(dmu_bonus_hold(os, obj, FTAG, &db));
76     VERIFY3U(0, ==, dmu_bonus_hold(os, obj, FTAG, &db));
77     dmu_buf_will_dirty(db, tx);
78     bt = db->db_data;
79     bt->bt_begin = 0;
80     bt->bt_end = 0;
81     bt->bt_bytes = 0;
82     bt->bt_comp = 0;
83     bt->bt_uncomp = 0;
84     dmu_buf_rele(db, FTAG);

85     return (obj);
86 }

88 int
89 bptree_free(objset_t *os, uint64_t obj, dmu_tx_t *tx)
90 {
91     dmu_buf_t *db;
92     bptree_phys_t *bt;

94     VERIFY0(dmu_bonus_hold(os, obj, FTAG, &db));
95     VERIFY3U(0, ==, dmu_bonus_hold(os, obj, FTAG, &db));
96     bt = db->db_data;
97     ASSERT3U(bt->bt_begin, ==, bt->bt_end);
98     ASSERT0(bt->bt_bytes);
99     ASSERT0(bt->bt_comp);
100    ASSERT0(bt->bt_uncomp);
101    ASSERT3U(bt->bt_bytes, ==, 0);
102    ASSERT3U(bt->bt_comp, ==, 0);
103    ASSERT3U(bt->bt_uncomp, ==, 0);
104    dmu_buf_rele(db, FTAG);

105    return (dmu_object_free(os, obj, tx));
106 }

107 void
108 bptree_add(objset_t *os, uint64_t obj, blkptr_t *bp, uint64_t birth_txg,
109     uint64_t bytes, uint64_t comp, uint64_t uncomp, dmu_tx_t *tx)
110 {
111     dmu_buf_t *db;
112     bptree_phys_t *bt;
113     bptree_entry_phys_t bte;
114 }

```

```

114     * bptree objects are in the pool mos, therefore they can only be
115     * modified in syncing context. Furthermore, this is only modified
116     * by the sync thread, so no locking is necessary.
117     */
118     ASSERT(dmu_tx_is_syncing(tx));

120     VERIFY0(dmu_bonus_hold(os, obj, FTAG, &db));
121     VERIFY3U(0, ==, dmu_bonus_hold(os, obj, FTAG, &db));
122     bt = db->db_data;

123     bte.be_birth_txg = birth_txg;
124     bte.be_bp = *bp;
125     bzero(&bte.be_zb, sizeof (bte.be_zb));
126     dmu_write(os, obj, bt->bt_end * sizeof (bte), sizeof (bte), &bte, tx);

128     dmu_buf_will_dirty(db, tx);
129     bt->bt_end++;
130     bt->bt_bytes += bytes;
131     bt->bt_comp += comp;
132     bt->bt_uncomp += uncomp;
133     dmu_buf_rele(db, FTAG);
134 }
_____unchanged_portion_omitted_____

156 int
157 bptree_iterate(objset_t *os, uint64_t obj, boolean_t free, bptree_itor_t func,
158     void *arg, dmu_tx_t *tx)
159 {
160     int err;
161     uint64_t i;
162     dmu_buf_t *db;
163     struct bptree_args ba;

165     ASSERT(!free || dmu_tx_is_syncing(tx));

167     err = dmu_bonus_hold(os, obj, FTAG, &db);
168     if (err != 0)
169         return (err);

171     if (free)
172         dmu_buf_will_dirty(db, tx);

174     ba.ba_phys = db->db_data;
175     ba.ba_free = free;
176     ba.ba_func = func;
177     ba.ba_arg = arg;
178     ba.ba_tx = tx;

180     err = 0;
181     for (i = ba.ba_phys->bt_begin; i < ba.ba_phys->bt_end; i++) {
182         bptree_entry_phys_t bte;

184         ASSERT(!free || i == ba.ba_phys->bt_begin);

186         err = dmu_read(os, obj, i * sizeof (bte), sizeof (bte),
187             &bte, DMU_READ_NO_PREFETCH);
188         if (err != 0)
189             break;

191         err = traverse_dataset_destroyed(os->os_spa, &bte.be_bp,
192             bte.be_birth_txg, &bte.be_zb, TRAVERSE_POST,
193             bptree_visit_cb, &ba);
194         if (free) {
195             ASSERT(err == 0 || err == ERESTART);
196             if (err != 0) {
197                 /* save bookmark for future resume */

```

```
198         ASSERT3U(bte.be_zb.zb_objset, ==,  
199                 ZB_DESTROYED_OBJSET);  
200         ASSERT0(bte.be_zb.zb_level);  
200         ASSERT3U(bte.be_zb.zb_level, ==, 0);  
201         dmu_write(os, obj, i * sizeof (bte),  
202                 sizeof (bte), &bte, tx);  
203         break;  
204     } else {  
205         ba.ba_phys->bt_begin++;  
206         (void) dmu_free_range(os, obj,  
207                             i * sizeof (bte), sizeof (bte), tx);  
208     }  
209 }  
210  
212 ASSERT(!free || err != 0 || ba.ba_phys->bt_begin == ba.ba_phys->bt_end);  
  
214 /* if all blocks are free there should be no used space */  
215 if (ba.ba_phys->bt_begin == ba.ba_phys->bt_end) {  
216     ASSERT0(ba.ba_phys->bt_bytes);  
217     ASSERT0(ba.ba_phys->bt_comp);  
218     ASSERT0(ba.ba_phys->bt_uncomp);  
216     ASSERT3U(ba.ba_phys->bt_bytes, ==, 0);  
217     ASSERT3U(ba.ba_phys->bt_comp, ==, 0);  
218     ASSERT3U(ba.ba_phys->bt_uncomp, ==, 0);  
219 }  
  
221     dmu_buf_rele(db, FTAG);  
  
223     return (err);  
224 }  
  
unchanged_portion_omitted
```

```

*****
73444 Thu Jul 26 14:19:06 2012
new/usr/src/uts/common/fs/zfs/dbuf.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

294 /*
295  * Other stuff.
296  */

298 #ifdef ZFS_DEBUG
299 static void
300 dbuf_verify(dmu_buf_impl_t *db)
301 {
302     dnode_t *dn;
303     dbuf_dirty_record_t *dr;

305     ASSERT(MUTEX_HELD(&db->db_mtx));

307     if (!(zfs_flags & ZFS_DEBUG_DBUF_VERIFY))
308         return;

310     ASSERT(db->db_objset != NULL);
311     DB_DNODE_ENTER(db);
312     dn = DB_DNODE(db);
313     if (dn == NULL) {
314         ASSERT(db->db_parent == NULL);
315         ASSERT(db->db_blkptr == NULL);
316     } else {
317         ASSERT3U(db->db.db_object, ==, dn->dn_object);
318         ASSERT3P(db->db.objset, ==, dn->dn_objset);
319         ASSERT3U(db->db.level, <, dn->dn_nlevels);
320         ASSERT(db->db.blkid == DMU_BONUS_BLKID ||
321             db->db.blkid == DMU_SPILL_BLKID ||
322             !list_is_empty(&dn->dn_dbufs));
323     }
324     if (db->db.blkid == DMU_BONUS_BLKID) {
325         ASSERT(dn != NULL);
326         ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
327         ASSERT3U(db->db.db_offset, ==, DMU_BONUS_BLKID);
328     } else if (db->db.blkid == DMU_SPILL_BLKID) {
329         ASSERT(dn != NULL);
330         ASSERT3U(db->db.db_size, >=, dn->dn_bonuslen);
331         ASSERT0(db->db.db_offset);
332         ASSERT3U(db->db.db_offset, ==, 0);
333     } else {
334         ASSERT3U(db->db.db_offset, ==, db->db.blkid * db->db.db_size);
335     }

336     for (dr = db->db_data_pending; dr != NULL; dr = dr->dr_next)
337         ASSERT(dr->dr_dbuf == db);

339     for (dr = db->db_last_dirty; dr != NULL; dr = dr->dr_next)
340         ASSERT(dr->dr_dbuf == db);

342     /*
343     * We can't assert that db_size matches dn_datablksz because it
344     * can be momentarily different when another thread is doing
345     * dnode_set_blkisz().
346     */
347     if (db->db.level == 0 && db->db.db_object == DMU_META_DNODE_OBJECT) {
348         dr = db->db_data_pending;
349         /*
350         * It should only be modified in syncing context, so
351         * make sure we only have one copy of the data.

```

```

352     */
353     ASSERT(dr == NULL || dr->dt.dl.dr_data == db->db_buf);
354 }

356     /* verify db->db_blkptr */
357     if (db->db_blkptr) {
358         if (db->db_parent == dn->dn_dbuf) {
359             /* db is pointed to by the dnode */
360             /* ASSERT3U(db->db.blkid, <, dn->dn_nblkptr); */
361             if (DMU_OBJECT_IS_SPECIAL(db->db.db_object))
362                 ASSERT(db->db_parent == NULL);
363             else
364                 ASSERT(db->db_parent != NULL);
365             if (db->db.blkid != DMU_SPILL_BLKID)
366                 ASSERT3P(db->db_blkptr, ==,
367                     &dn->dn_phys->dn_blkptr[db->db.blkid]);
368         } else {
369             /* db is pointed to by an indirect block */
370             int epb = db->db_parent->db.db_size >> SPA_BLKPTRSHIFT;
371             ASSERT3U(db->db_parent->db.level, ==, db->db.level+1);
372             ASSERT3U(db->db_parent->db.db_object, ==,
373                 db->db.db_object);
374             /*
375             * dnode_grow_indblksz() can make this fail if we don't
376             * have the struct_rwlock. XXX indblksz no longer
377             * grows. safe to do this now?
378             */
379             if (RW_WRITE_HELD(&dn->dn_struct_rwlock)) {
380                 ASSERT3P(db->db_blkptr, ==,
381                     ((blkptr_t *)db->db_parent->db.db_data +
382                     db->db.blkid % epb));
383             }
384         }
385     }
386     if ((db->db_blkptr == NULL || BP_IS_HOLE(db->db_blkptr)) &&
387         (db->db_buf == NULL || db->db_buf->b_data) &&
388         db->db.db_data && db->db.blkid != DMU_BONUS_BLKID &&
389         db->db.state != DB_FILL && !dn->dn_free_txg) {
390         /*
391         * If the blkptr isn't set but they have nonzero data,
392         * it had better be dirty, otherwise we'll lose that
393         * data when we evict this buffer.
394         */
395         if (db->db_dirtycnt == 0) {
396             uint64_t *buf = db->db.db_data;
397             int i;

399             for (i = 0; i < db->db.db_size >> 3; i++) {
400                 ASSERT(buf[i] == 0);
401             }
402         }
403     }
404     DB_DNODE_EXIT(db);
405 }

_____unchanged_portion_omitted_____

2263 static void
2264 dbuf_sync_leaf(dbuf_dirty_record_t *dr, dmu_tx_t *tx)
2265 {
2266     arc_buf_t **datap = &dr->dt.dl.dr_data;
2267     dmu_buf_impl_t *db = dr->dr_dbuf;
2268     dnode_t *dn;
2269     objset_t *os;
2270     uint64_t txg = tx->tx_txg;

2272     ASSERT(dmu_tx_is_syncing(tx));

```

```

2274     dprintf(&dbuf_bp(db, db->db_blkptr, "blkptr=%p", db->db_blkptr);
2275
2276     mutex_enter(&db->db_mtx);
2277     /*
2278      * To be synced, we must be dirtied. But we
2279      * might have been freed after the dirty.
2280      */
2281     if (db->db_state == DB_UNCACHED) {
2282         /* This buffer has been freed since it was dirtied */
2283         ASSERT(db->db_data == NULL);
2284     } else if (db->db_state == DB_FILL) {
2285         /* This buffer was freed and is now being re-filled */
2286         ASSERT(db->db_data != dr->dt.dl.dr_data);
2287     } else {
2288         ASSERT(db->db_state == DB_CACHED || db->db_state == DB_NOFILL);
2289     }
2290     DBUF_VERIFY(db);
2291
2292     DB_DNODE_ENTER(db);
2293     dn = DB_DNODE(db);
2294
2295     if (db->db_blkid == DMU_SPILL_BLKID) {
2296         mutex_enter(&dn->dn_mtx);
2297         dn->dn_phys->dn_flags |= DNODE_FLAG_SPILL_BLKPTR;
2298         mutex_exit(&dn->dn_mtx);
2299     }
2300
2301     /*
2302      * If this is a bonus buffer, simply copy the bonus data into the
2303      * dnode. It will be written out when the dnode is synced (and it
2304      * will be synced, since it must have been dirty for dbuf_sync to
2305      * be called).
2306      */
2307     if (db->db_blkid == DMU_BONUS_BLKID) {
2308         dbuf_dirty_record_t **drp;
2309
2310         ASSERT(*datap != NULL);
2311         ASSERT0(db->db_level);
2312         ASSERT3U(db->db_level, ==, 0);
2313         ASSERT3U(dn->dn_phys->dn_bonuslen, <=, DN_MAX_BONUSLEN);
2314         bcopy(*datap, DN_BONUS(dn->dn_phys), dn->dn_phys->dn_bonuslen);
2315         DB_DNODE_EXIT(db);
2316
2317         if (*datap != db->db_data) {
2318             zio_buf_free(*datap, DN_MAX_BONUSLEN);
2319             arc_space_return(DN_MAX_BONUSLEN, ARC_SPACE_OTHER);
2320         }
2321         db->db_data_pending = NULL;
2322         drp = &db->db_last_dirty;
2323         while (*drp != dr)
2324             drp = &(*drp)->dr_next;
2325         ASSERT(dr->dr_next == NULL);
2326         ASSERT(dr->dr_dbuf == db);
2327         *drp = dr->dr_next;
2328         kmem_free(dr, sizeof (dbuf_dirty_record_t));
2329         ASSERT(db->db_dirtycnt > 0);
2330         db->db_dirtycnt -= 1;
2331         dbuf_rele_and_unlock(db, (void *) (uintptr_t)txg);
2332         return;
2333     }
2334
2335     os = dn->dn_objset;
2336
2337     /*
2338      * This function may have dropped the db_mtx lock allowing a dmu_sync

```

```

2338     * operation to sneak in. As a result, we need to ensure that we
2339     * don't check the dr_override_state until we have returned from
2340     * dbuf_check_blkptr.
2341     */
2342     dbuf_check_blkptr(dn, db);
2343
2344     /*
2345      * If this buffer is in the middle of an immediate write,
2346      * wait for the synchronous IO to complete.
2347      */
2348     while (dr->dt.dl.dr_override_state == DR_IN_DMU_SYNC) {
2349         ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
2350         cv_wait(&db->db_changed, &db->db_mtx);
2351         ASSERT(dr->dt.dl.dr_override_state != DR_NOT_OVERRIDDEN);
2352     }
2353
2354     if (db->db_state != DB_NOFILL &&
2355         dn->dn_object != DMU_META_DNODE_OBJECT &&
2356         refcount_count(&db->db_holds) > 1 &&
2357         dr->dt.dl.dr_override_state != DR_OVERRIDDEN &&
2358         *datap == db->db_buf) {
2359         /*
2360          * If this buffer is currently "in use" (i.e., there
2361          * are active holds and db_data still references it),
2362          * then make a copy before we start the write so that
2363          * any modifications from the open txg will not leak
2364          * into this write.
2365          *
2366          * NOTE: this copy does not need to be made for
2367          * objects only modified in the syncing context (e.g.
2368          * DNODE_DNODE blocks).
2369          */
2370         int blkksz = arc_buf_size(*datap);
2371         arc_buf_contents_t type = DBUF_GET_BUFC_TYPE(db);
2372         *datap = arc_buf_alloc(os->os_spa, blkksz, db, type);
2373         bcopy(db->db_data, (*datap)->b_data, blkksz);
2374     }
2375     db->db_data_pending = dr;
2376
2377     mutex_exit(&db->db_mtx);
2378
2379     dbuf_write(dr, *datap, tx);
2380
2381     ASSERT(!list_link_active(&dr->dr_dirty_node));
2382     if (dn->dn_object == DMU_META_DNODE_OBJECT) {
2383         list_insert_tail(&dn->dn_dirty_records[txg&TXG_MASK], dr);
2384         DB_DNODE_EXIT(db);
2385     } else {
2386         /*
2387          * Although zio_nowait() does not "wait for an IO", it does
2388          * initiate the IO. If this is an empty write it seems plausible
2389          * that the IO could actually be completed before the nowait
2390          * returns. We need to DB_DNODE_EXIT() first in case
2391          * zio_nowait() invalidates the dbuf.
2392          */
2393         DB_DNODE_EXIT(db);
2394         zio_nowait(dr->dr_zio);
2395     }
2396 }
2397
2398     unchanged_portion_omitted
2399
2400     /* ARGSUSED */
2401     static void
2402     dbuf_write_done(zio_t *zio, arc_buf_t *buf, void *vdb)
2403     {
2404         dmu_buf_impl_t *db = vdb;

```



```

2505     blkptr_t *bp = zio->io_bp;
2506     blkptr_t *bp_orig = &zio->io_bp_orig;
2507     uint64_t txg = zio->io_txg;
2508     dbuf_dirty_record_t **drp, *dr;

2510     ASSERT0(zio->io_error);
2510     ASSERT3U(zio->io_error, ==, 0);
2511     ASSERT(db->db_blkptr == bp);

2513     if (zio->io_flags & ZIO_FLAG_IO_REWRITE) {
2514         ASSERT(BP_EQUAL(bp, bp_orig));
2515     } else {
2516         objset_t *os;
2517         dsl_dataset_t *ds;
2518         dmu_tx_t *tx;

2520         DB_GET_OBJSET(&os, db);
2521         ds = os->os_dsl_dataset;
2522         tx = os->os_synctx;

2524         (void) dsl_dataset_block_kill(ds, bp_orig, tx, B_TRUE);
2525         dsl_dataset_block_born(ds, bp, tx);
2526     }

2528     mutex_enter(&db->db_mtx);

2530     DBUF_VERIFY(db);

2532     drp = &db->db_last_dirty;
2533     while ((dr = *drp) != db->db_data_pending)
2534         drp = &dr->dr_next;
2535     ASSERT(!list_link_active(&dr->dr_dirty_node));
2536     ASSERT(dr->dr_txg == txg);
2537     ASSERT(dr->dr_dbuf == db);
2538     ASSERT(dr->dr_next == NULL);
2539     *drp = dr->dr_next;

2541 #ifdef ZFS_DEBUG
2542     if (db->db_blkid == DMU_SPILL_BLKID) {
2543         dnode_t *dn;

2545         DB_DNODE_ENTER(db);
2546         dn = DB_DNODE(db);
2547         ASSERT(dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR);
2548         ASSERT(!(BP_IS_HOLE(db->db_blkptr)) &&
2549             db->db_blkptr == &dn->dn_phys->dn_spill);
2550         DB_DNODE_EXIT(db);
2551     }
2552 #endif

2554     if (db->db_level == 0) {
2555         ASSERT(db->db_blkid != DMU_BONUS_BLKID);
2556         ASSERT(dr->dt.dl.dr_override_state == DR_NOT_OVERRIDDEN);
2557         if (db->db_state != DB_NOFILL) {
2558             if (dr->dt.dl.dr_data != db->db_buf)
2559                 VERIFY(arc_buf_remove_ref(dr->dt.dl.dr_data,
2560                     db) == 1);
2561             else if (!arc_released(db->db_buf))
2562                 arc_set_callback(db->db_buf, dbuf_do_evict, db);
2563         }
2564     } else {
2565         dnode_t *dn;

2567         DB_DNODE_ENTER(db);
2568         dn = DB_DNODE(db);
2569         ASSERT(list_head(&dr->dt.di.dr_children) == NULL);

```

```

2570         ASSERT3U(db->db_size, ==, 1<<dn->dn_phys->dn_indblkshift);
2571         if (!BP_IS_HOLE(db->db_blkptr)) {
2572             int epbs =
2573                 dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
2574             ASSERT3U(BP_GET_LSIZE(db->db_blkptr), ==,
2575                 db->db_size);
2576             ASSERT3U(dn->dn_phys->dn_maxblkid
2577                 >> (db->db_level * epbs), >=, db->db_blkid);
2578             arc_set_callback(db->db_buf, dbuf_do_evict, db);
2579         }
2580         DB_DNODE_EXIT(db);
2581         mutex_destroy(&dr->dt.di.dr_mtx);
2582         list_destroy(&dr->dt.di.dr_children);
2583     }
2584     kmem_free(dr, sizeof (dbuf_dirty_record_t));

2586     cv_broadcast(&db->db_changed);
2587     ASSERT(db->db_dirtycnt > 0);
2588     db->db_dirtycnt -= 1;
2589     db->db_data_pending = NULL;
2590     dbuf_rele_and_unlock(db, (void *) (uintptr_t)txg);
2591 }

```

unchanged portion omitted

```

*****
45183 Thu Jul 26 14:19:07 2012
new/usr/src/uts/common/fs/zfs/dmu_objset.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

696 static void
697 dmu_objset_create_sync(void *arg1, void *arg2, dmu_tx_t *tx)
698 {
699     dsl_dir_t *dd = arg1;
700     spa_t *spa = dd->dd_pool->dp_spa;
701     struct oscarg *oa = arg2;
702     uint64_t obj;
703     dsl_dataset_t *ds;
704     blkptr_t *bp;

706     ASSERT(dmu_tx_is_syncing(tx));

708     obj = dsl_dataset_create_sync(dd, oa->lastname,
709     oa->clone_origin, oa->flags, oa->cr, tx);

711     VERIFY0(dsl_dataset_hold_obj(dd->dd_pool, obj, FTAG, &ds));
711     VERIFY3U(0, ==, dsl_dataset_hold_obj(dd->dd_pool, obj, FTAG, &ds));
712     bp = dsl_dataset_get_blkptr(ds);
713     if (BP_IS_HOLE(bp)) {
714         objset_t *os =
715             dmu_objset_create_impl(spa, ds, bp, oa->type, tx);

717         if (oa->userfunc)
718             oa->userfunc(os, oa->userarg, oa->cr, tx);
719     }

721     if (oa->clone_origin == NULL) {
722         spa_history_log_internal_ds(ds, "create", tx, "");
723     } else {
724         char namebuf[MAXNAMELEN];
725         dsl_dataset_name(oa->clone_origin, namebuf);
726         spa_history_log_internal_ds(ds, "clone", tx,
727             "origin=%s (%llu)", namebuf, oa->clone_origin->ds_object);
728     }
729     dsl_dataset_rele(ds, FTAG);
730 }
_____unchanged_portion_omitted_____

1143 /* called from dsl */
1144 void
1145 dmu_objset_sync(objset_t *os, zio_t *pio, dmu_tx_t *tx)
1146 {
1147     int txgoff;
1148     zbookmark_t zb;
1149     zio_prop_t zp;
1150     zio_t *zio;
1151     list_t *list;
1152     list_t *newlist = NULL;
1153     dbuf_dirty_record_t *dr;

1155     dprintf_ds(os->os_dsl_dataset, "txg=%llu\n", tx->tx_txg);

1157     ASSERT(dmu_tx_is_syncing(tx));
1158     /* XXX the write_done callback should really give us the tx... */
1159     os->os_synctx = tx;

1161     if (os->os_dsl_dataset == NULL) {
1162         /*
1163          * This is the MOS. If we have upgraded,

```

```

1164         * spa_max_replication() could change, so reset
1165         * os_copies here.
1166         */
1167         os->os_copies = spa_max_replication(os->os_spa);
1168     }

1170     /*
1171     * Create the root block IO
1172     */
1173     SET_BOOKMARK(&zb, os->os_dsl_dataset ?
1174         os->os_dsl_dataset->ds_object : DMU_META_OBJSET,
1175         ZB_ROOT_OBJECT, ZB_ROOT_LEVEL, ZB_ROOT_BLKID);
1176     VERIFY0(arc_release_bp(os->os_phys_buf, &os->os_phys_buf,
1177     VERIFY3U(0, ==, arc_release_bp(os->os_phys_buf, &os->os_phys_buf,
1178         os->os_rootbp, os->os_spa, &zb));

1179     dmu_write_policy(os, NULL, 0, 0, &zp);

1181     zio = arc_write(pio, os->os_spa, tx->tx_txg,
1182         os->os_rootbp, os->os_phys_buf, DMU_OS_IS_L2CACHEABLE(os), &zp,
1183         dmu_objset_write_ready, dmu_objset_write_done, os,
1184         ZIO_PRIORITY_ASYNC_WRITE, ZIO_FLAG_MUSTSUCCEED, &zb);

1186     /*
1187     * Sync special dnodes - the parent IO for the sync is the root block
1188     */
1189     DMU_META_DNODE(os)->dn_zio = zio;
1190     dnode_sync(DMU_META_DNODE(os), tx);

1192     os->os_phys->os_flags = os->os_flags;

1194     if (DMU_USERUSED_DNODE(os) &&
1195         DMU_USERUSED_DNODE(os)->dn_type != DMU_OT_NONE) {
1196         DMU_USERUSED_DNODE(os)->dn_zio = zio;
1197         dnode_sync(DMU_USERUSED_DNODE(os), tx);
1198         DMU_GROUPUSED_DNODE(os)->dn_zio = zio;
1199         dnode_sync(DMU_GROUPUSED_DNODE(os), tx);
1200     }

1202     txgoff = tx->tx_txg & TXG_MASK;

1204     if (dmu_objset_userused_enabled(os)) {
1205         newlist = &os->os_synced_dnodes;
1206         /*
1207         * We must create the list here because it uses the
1208         * dn_dirty_link[] of this txg.
1209         */
1210         list_create(newlist, sizeof (dnode_t),
1211             offsetof(dnode_t, dn_dirty_link[txgoff]));
1212     }

1214     dmu_objset_sync_dnodes(&os->os_free_dnodes[txgoff], newlist, tx);
1215     dmu_objset_sync_dnodes(&os->os_dirty_dnodes[txgoff], newlist, tx);

1217     list = &DMU_META_DNODE(os)->dn_dirty_records[txgoff];
1218     while (dr = list_head(list)) {
1219         ASSERT(dr->dr_dbuf->db_level == 0);
1220         list_remove(list, dr);
1221         if (dr->dr_zio)
1222             zio_nwait(dr->dr_zio);
1223     }
1224     /*
1225     * Free intent log blocks up to this tx.
1226     */
1227     zil_sync(os->os_zil, tx);
1228     os->os_phys->os_zil_header = os->os_zil_header;

```

```
1229         zio_nowait(zio);
1230     }
    _____unchanged_portion_omitted_

1264 static void
1265 do_userquota_update(objset_t *os, uint64_t used, uint64_t flags,
1266     uint64_t user, uint64_t group, boolean_t subtract, dmu_tx_t *tx)
1267 {
1268     if ((flags & DNODE_FLAG_USERUSED_ACCOUNTED)) {
1269         int64_t delta = DNODE_SIZE + used;
1270         if (subtract)
1271             delta = -delta;
1272         VERIFY0(zap_increment_int(os, DMU_USERUSED_OBJECT,
1272             VERIFY3U(0, ==, zap_increment_int(os, DMU_USERUSED_OBJECT,
1273                 user, delta, tx));
1274         VERIFY0(zap_increment_int(os, DMU_GROUPUSED_OBJECT,
1274             VERIFY3U(0, ==, zap_increment_int(os, DMU_GROUPUSED_OBJECT,
1275                 group, delta, tx));
1276     }
1277 }
    _____unchanged_portion_omitted_
```

```

*****
45439 Thu Jul 26 14:19:08 2012
new/usr/src/uts/common/fs/zfs/dmu_send.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 2011 Nexenta Systems, Inc. All rights reserved.
24 * Copyright (c) 2012 by Delphix. All rights reserved.
25 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
26 */

28 #include <sys/dmu.h>
29 #include <sys/dmu_impl.h>
30 #include <sys/dmu_tx.h>
31 #include <sys/dbuf.h>
32 #include <sys/dnode.h>
33 #include <sys/zfs_context.h>
34 #include <sys/dmu_objset.h>
35 #include <sys/dmu_traverse.h>
36 #include <sys/dsl_dataset.h>
37 #include <sys/dsl_dir.h>
38 #include <sys/dsl_prop.h>
39 #include <sys/dsl_pool.h>
40 #include <sys/dsl_synctask.h>
41 #include <sys/zfs_ioctl.h>
42 #include <sys/zap.h>
43 #include <sys/zio_checksum.h>
44 #include <sys/zfs_znode.h>
45 #include <zfs_fletcher.h>
46 #include <sys/avl.h>
47 #include <sys/ddt.h>
48 #include <sys/zfs_onexit.h>

50 /* Set this tunable to TRUE to replace corrupt data with 0x2f5baddb10c */
51 int zfs_send_corrupt_data = B_FALSE;

53 static char *dmu_recv_tag = "dmu_recv_tag";

55 static int
56 dump_bytes(dmu_sendarg_t *dsp, void *buf, int len)
57 {
58     dsl_dataset_t *ds = dsp->dsa_os->os_dsl_dataset;
59     ssize_t resid; /* have to get resid to get detailed errno */
60     ASSERT0(len % 8);
61     ASSERT3U(len % 8, ==, 0);

```

```

62     fletcher_4_incremental_native(buf, len, &dsp->dsa_zc);
63     dsp->dsa_err = vn_rdwr(UIO_WRITE, dsp->dsa_vp,
64         (caddr_t)buf, len,
65         0, UIO_SYSSPACE, FAPPEND, RLIM64_INFINITY, CRED(), &resid);

67     mutex_enter(&ds->ds_sendstream_lock);
68     *dsp->dsa_off += len;
69     mutex_exit(&ds->ds_sendstream_lock);

71     return (dsp->dsa_err);
72 }

unchanged_portion_omitted

957 static void *
958 restore_read(struct restorearg *ra, int len)
959 {
960     void *rv;
961     int done = 0;

963     /* some things will require 8-byte alignment, so everything must */
964     ASSERT0(len % 8);
964     ASSERT3U(len % 8, ==, 0);

966     while (done < len) {
967         ssize_t resid;

969         ra->err = vn_rdwr(UIO_READ, ra->vp,
970             (caddr_t)ra->buf + done, len - done,
971             ra->voff, UIO_SYSSPACE, FAPPEND,
972             RLIM64_INFINITY, CRED(), &resid);

974         if (resid == len - done)
975             ra->err = EINVAL;
976         ra->voff += len - done - resid;
977         done = len - resid;
978         if (ra->err)
979             return (NULL);
980     }

982     ASSERT3U(done, ==, len);
983     rv = ra->buf;
984     if (ra->byteswap)
985         fletcher_4_incremental_byteswap(rv, len, &ra->cksum);
986     else
987         fletcher_4_incremental_native(rv, len, &ra->cksum);
988     return (rv);
989 }

unchanged_portion_omitted

1604 static int
1605 dmu_recv_existing_end(dmu_recv_cookie_t *drc)
1606 {
1607     struct recvendsyncarg resa;
1608     dsl_dataset_t *ds = drc->drc_logical_ds;
1609     int err, myerr;

1611     /*
1612     * XXX hack; seems the ds is still dirty and dsl_pool_zil_clean()
1613     * expects it to have a ds_user_ptr (and zil), but clone_swap()
1614     * can close it.
1615     */
1616     txg_wait_synced(ds->ds_dir->dd_pool, 0);

1618     if (dsl_dataset_tryown(ds, FALSE, dmu_recv_tag)) {
1619         err = dsl_dataset_clone_swap(drc->drc_real_ds, ds,

```

```
1620         drc->drc_force);
1621         if (err)
1622             goto out;
1623     } else {
1624         mutex_exit(&ds->ds_recvlock);
1625         dsl_dataset_rele(ds, dmu_recv_tag);
1626         (void) dsl_dataset_destroy(drc->drc_real_ds, dmu_recv_tag,
1627             B_FALSE);
1628         return (EBUSY);
1629     }
1631     resa.creation_time = drc->drc_drrb->drr_creation_time;
1632     resa.toguid = drc->drc_drrb->drr_toguid;
1633     resa.tosnap = drc->drc_tosnap;
1635     err = dsl_sync_task_do(ds->ds_dir->dd_pool,
1636         recv_end_check, recv_end_sync, ds, &resa, 3);
1637     if (err) {
1638         /* swap back */
1639         (void) dsl_dataset_clone_swap(drc->drc_real_ds, ds, B_TRUE);
1640     }
1642 out:
1643     mutex_exit(&ds->ds_recvlock);
1644     if (err == 0 && drc->drc_guid_to_ds_map != NULL)
1645         (void) add_ds_to_guidmap(drc->drc_guid_to_ds_map, ds);
1646     dsl_dataset_disown(ds, dmu_recv_tag);
1647     myerr = dsl_dataset_destroy(drc->drc_real_ds, dmu_recv_tag, B_FALSE);
1648     ASSERT0(myerr);
1648     ASSERT3U(myerr, ==, 0);
1649     return (err);
1650 }
_____unchanged_portion_omitted_____
```

new/usr/src/uts/common/fs/zfs/dmu\_traverse.c

1

\*\*\*\*\*

14478 Thu Jul 26 14:19:09 2012

new/usr/src/uts/common/fs/zfs/dmu\_traverse.c

3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero

\*\*\*\*\*

\_\_\_\_\_unchanged\_portion\_omitted\_

```
173 static void
174 traverse_pause(traverse_data_t *td, const zbookmark_t *zb)
175 {
176     ASSERT(td->td_resume != NULL);
177     ASSERT0(zb->zb_level);
177     ASSERT3U(zb->zb_level, ==, 0);
178     bcopy(zb, td->td_resume, sizeof (*td->td_resume));
179 }
```

\_\_\_\_\_unchanged\_portion\_omitted\_

```

*****
35009 Thu Jul 26 14:19:10 2012
new/usr/src/uts/common/fs/zfs/dmu_tx.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____
892 #endif

894 static int
895 dmu_tx_try_assign(dmu_tx_t *tx, uint64_t txg_how)
896 {
897     dmu_tx_hold_t *txh;
898     spa_t *spa = tx->tx_pool->dp_spa;
899     uint64_t memory, asize, fsize, usize;
900     uint64_t towrite, tofree, tooverwrite, tounref, tohold, fudge;

902     ASSERT0(tx->tx_txg);
902     ASSERT3U(tx->tx_txg, ==, 0);

904     if (tx->tx_err)
905         return (tx->tx_err);

907     if (spa_suspended(spa)) {
908         /*
909          * If the user has indicated a blocking failure mode
910          * then return ERESTART which will block in dmu_tx_wait().
911          * Otherwise, return EIO so that an error can get
912          * propagated back to the VOP calls.
913          */
914         /* Note that we always honor the txg_how flag regardless
915          * of the failuremode setting.
916          */
917         if (spa_get_failmode(spa) == ZIO_FAILURE_MODE_CONTINUE &&
918             txg_how != TXG_WAIT)
919             return (EIO);

921         return (ERESTART);
922     }

924     tx->tx_txg = txg_hold_open(tx->tx_pool, &tx->tx_txgh);
925     tx->tx_needassign_txh = NULL;

927     /*
928      * NB: No error returns are allowed after txg_hold_open, but
929      * before processing the dnode holds, due to the
930      * dmu_tx_unassign() logic.
931      */

933     towrite = tofree = tooverwrite = tounref = tohold = fudge = 0;
934     for (txh = list_head(&tx->tx_holds); txh;
935          txh = list_next(&tx->tx_holds, txh)) {
936         dnode_t *dn = txh->txh_dnode;
937         if (dn != NULL) {
938             mutex_enter(&dn->dn_mtx);
939             if (dn->dn_assigned_txg == tx->tx_txg - 1) {
940                 mutex_exit(&dn->dn_mtx);
941                 tx->tx_needassign_txh = txh;
942                 return (ERESTART);
943             }
944             if (dn->dn_assigned_txg == 0)
945                 dn->dn_assigned_txg = tx->tx_txg;
946             ASSERT3U(dn->dn_assigned_txg, ==, tx->tx_txg);
947             (void) refcount_add(&dn->dn_tx_holds, tx);
948             mutex_exit(&dn->dn_mtx);
949         }
950         towrite += txh->txh_space_towrite;

```

```

951         tofree += txh->txh_space_tofree;
952         tooverwrite += txh->txh_space_tooverwrite;
953         tounref += txh->txh_space_tounref;
954         tohold += txh->txh_memory_tohold;
955         fudge += txh->txh_fudge;
956     }

958     /*
959      * NB: This check must be after we've held the dnodes, so that
960      * the dmu_tx_unassign() logic will work properly
961      */
962     if (txg_how >= TXG_INITIAL && txg_how != tx->tx_txg)
963         return (ERESTART);

965     /*
966      * If a snapshot has been taken since we made our estimates,
967      * assume that we won't be able to free or overwrite anything.
968      */
969     if (tx->tx_objset &&
970         dsl_dataset_prev_snap_txg(tx->tx_objset->os_dsl_dataset) >
971         tx->tx_lastsnap_txg) {
972         towrite += tooverwrite;
973         tooverwrite = tofree = 0;
974     }

976     /* needed allocation: worst-case estimate of write space */
977     asize = spa_get_asize(tx->tx_pool->dp_spa, towrite + tooverwrite);
978     /* freed space estimate: worst-case overwrite + free estimate */
979     fsize = spa_get_asize(tx->tx_pool->dp_spa, tooverwrite) + tofree;
980     /* convert unref'd space to worst-case estimate */
981     usize = spa_get_asize(tx->tx_pool->dp_spa, tounref);
982     /* calculate memory footprint estimate */
983     memory = towrite + tooverwrite + tohold;

985 #ifdef ZFS_DEBUG
986     /*
987      * Add in 'tohold' to account for our dirty holds on this memory
988      * XXX - the "fudge" factor is to account for skipped blocks that
989      * we missed because dnode_next_offset() misses in-core-only blocks.
990      */
991     tx->tx_space_towrite = asize +
992         spa_get_asize(tx->tx_pool->dp_spa, tohold + fudge);
993     tx->tx_space_tofree = tofree;
994     tx->tx_space_tooverwrite = tooverwrite;
995     tx->tx_space_tounref = tounref;
996 #endif

998     if (tx->tx_dir && asize != 0) {
999         int err = dsl_dir_temppreserve_space(tx->tx_dir, memory,
1000             asize, fsize, usize, &tx->tx_temppreserve_cookie, tx);
1001         if (err)
1002             return (err);
1003     }

1005     return (0);
1006 }
_____unchanged_portion_omitted_____

```

```
*****
56217 Thu Jul 26 14:19:11 2012
new/usr/src/uts/common/fs/zfs/dnode.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
```

unchanged portion omitted

```
121 /* ARGSUSED */
122 static void
123 dnode_dest(void *arg, void *unused)
124 {
125     int i;
126     dnode_t *dn = arg;

128     rw_destroy(&dn->dn_struct_rwlock);
129     mutex_destroy(&dn->dn_mtx);
130     mutex_destroy(&dn->dn_dbufs_mtx);
131     cv_destroy(&dn->dn_notxholds);
132     refcount_destroy(&dn->dn_holds);
133     refcount_destroy(&dn->dn_tx_holds);
134     ASSERT(!list_link_active(&dn->dn_link));

136     for (i = 0; i < TXG_SIZE; i++) {
137         ASSERT(!list_link_active(&dn->dn_dirty_link[i]));
138         avl_destroy(&dn->dn_ranges[i]);
139         list_destroy(&dn->dn_dirty_records[i]);
140         ASSERT0(dn->dn_next_nblkptr[i]);
141         ASSERT0(dn->dn_next_nlevels[i]);
142         ASSERT0(dn->dn_next_indblkshift[i]);
143         ASSERT0(dn->dn_next_bonustype[i]);
144         ASSERT0(dn->dn_rm_spillblk[i]);
145         ASSERT0(dn->dn_next_bonuslen[i]);
146         ASSERT0(dn->dn_next_blkisz[i]);
147         ASSERT3U(dn->dn_next_nblkptr[i], ==, 0);
148         ASSERT3U(dn->dn_next_nlevels[i], ==, 0);
149         ASSERT3U(dn->dn_next_indblkshift[i], ==, 0);
150         ASSERT3U(dn->dn_next_bonustype[i], ==, 0);
151         ASSERT3U(dn->dn_rm_spillblk[i], ==, 0);
152         ASSERT3U(dn->dn_next_bonuslen[i], ==, 0);
153         ASSERT3U(dn->dn_next_blkisz[i], ==, 0);
154     }

149     ASSERT0(dn->dn_allocated_tgx);
150     ASSERT0(dn->dn_free_tgx);
151     ASSERT0(dn->dn_assigned_tgx);
152     ASSERT0(dn->dn_dirtyctx);
153     ASSERT3U(dn->dn_allocated_tgx, ==, 0);
154     ASSERT3U(dn->dn_free_tgx, ==, 0);
155     ASSERT3U(dn->dn_assigned_tgx, ==, 0);
156     ASSERT3U(dn->dn_dirtyctx, ==, 0);
157     ASSERT3P(dn->dn_dirtyctx_firstset, ==, NULL);
158     ASSERT3P(dn->dn_bonus, ==, NULL);
159     ASSERT(!dn->dn_have_spill);
160     ASSERT3P(dn->dn_zio, ==, NULL);
161     ASSERT0(dn->dn_oldused);
162     ASSERT0(dn->dn_oldflags);
163     ASSERT0(dn->dn_olduid);
164     ASSERT0(dn->dn_oldgid);
165     ASSERT0(dn->dn_newuid);
166     ASSERT0(dn->dn_newgid);
167     ASSERT0(dn->dn_id_flags);
168     ASSERT3U(dn->dn_oldused, ==, 0);
169     ASSERT3U(dn->dn_oldflags, ==, 0);
170     ASSERT3U(dn->dn_olduid, ==, 0);
171     ASSERT3U(dn->dn_oldgid, ==, 0);
172     ASSERT3U(dn->dn_newuid, ==, 0);
173     ASSERT3U(dn->dn_newgid, ==, 0);
```

```
162     ASSERT3U(dn->dn_newgid, ==, 0);
163     ASSERT3U(dn->dn_id_flags, ==, 0);
```

```
165     ASSERT0(dn->dn_dbufs_count);
166     ASSERT3U(dn->dn_dbufs_count, ==, 0);
167     list_destroy(&dn->dn_dbufs);
168 }
```

unchanged portion omitted

```
361 static void
362 dnode_setdblksz(dnode_t *dn, int size)
363 {
364     ASSERT0(P2PHASE(size, SPA_MINBLOCKSIZE));
365     ASSERT3U(P2PHASE(size, SPA_MINBLOCKSIZE), ==, 0);
366     ASSERT3U(size, <=, SPA_MAXBLOCKSIZE);
367     ASSERT3U(size, >=, SPA_MINBLOCKSIZE);
368     ASSERT3U(size >> SPA_MINBLOCKSHIFT, <,
369             1 << (sizeof (dn->dn_phys->dn_datablkzsec) * 8));
370     dn->dn_datablkzsec = size;
371     dn->dn_datablkzsec = size >> SPA_MINBLOCKSHIFT;
372     dn->dn_datablkshift = ISP2(size) ? highbit(size - 1) : 0;
373 }
```

unchanged portion omitted

```
477 void
478 dnode_allocate(dnode_t *dn, dmu_object_type_t ot, int blocksize, int ibs,
479             dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx)
480 {
481     int i;

483     if (blocksize == 0)
484         blocksize = 1 << zfs_default_bs;
485     else if (blocksize > SPA_MAXBLOCKSIZE)
486         blocksize = SPA_MAXBLOCKSIZE;
487     else
488         blocksize = P2ROUNDUP(blocksize, SPA_MINBLOCKSIZE);

490     if (ibs == 0)
491         ibs = zfs_default_ibs;

493     ibs = MIN(MAX(ibs, DN_MIN_INDBLKSHIFT), DN_MAX_INDBLKSHIFT);

495     dprintf("os=%p obj=%llu txg=%llu blocksize=%d ibs=%d\n", dn->dn_objset,
496             dn->dn_object, tx->tx_txg, blocksize, ibs);

498     ASSERT(dn->dn_type == DMU_OT_NONE);
499     ASSERT(bcmp(dn->dn_phys, &dnode_phys_zero, sizeof (dnode_phys_t)) == 0);
500     ASSERT(dn->dn_phys->dn_type == DMU_OT_NONE);
501     ASSERT(ot != DMU_OT_NONE);
502     ASSERT(DMU_OT_IS_VALID(ot));
503     ASSERT((bonustype == DMU_OT_NONE && bonuslen == 0) ||
504            (bonustype == DMU_OT_SA && bonuslen == 0) ||
505            (bonustype != DMU_OT_NONE && bonuslen != 0));
506     ASSERT(DMU_OT_IS_VALID(bonustype));
507     ASSERT3U(bonuslen, <=, DN_MAX_BONUSLEN);
508     ASSERT(dn->dn_type == DMU_OT_NONE);
509     ASSERT0(dn->dn_maxblkid);
510     ASSERT0(dn->dn_allocated_tgx);
511     ASSERT0(dn->dn_assigned_tgx);
512     ASSERT3U(dn->dn_maxblkid, ==, 0);
513     ASSERT3U(dn->dn_allocated_tgx, ==, 0);
514     ASSERT3U(dn->dn_assigned_tgx, ==, 0);
515     ASSERT(refcount_is_zero(&dn->dn_tx_holds));
516     ASSERT3U(refcount_count(&dn->dn_holds), <=, 1);
517     ASSERT3P(list_head(&dn->dn_dbufs), ==, NULL);
```



```

516     for (i = 0; i < TXG_SIZE; i++) {
517         ASSERT0(dn->dn_next_nblkptr[i]);
518         ASSERT0(dn->dn_next_nlevels[i]);
519         ASSERT0(dn->dn_next_indblkshift[i]);
520         ASSERT0(dn->dn_next_bonuslen[i]);
521         ASSERT0(dn->dn_next_bonustype[i]);
522         ASSERT0(dn->dn_rm_spillblk[i]);
523         ASSERT0(dn->dn_next_blkisz[i]);
524         ASSERT3U(dn->dn_next_nblkptr[i], ==, 0);
525         ASSERT3U(dn->dn_next_nlevels[i], ==, 0);
526         ASSERT3U(dn->dn_next_indblkshift[i], ==, 0);
527         ASSERT3U(dn->dn_next_bonuslen[i], ==, 0);
528         ASSERT3U(dn->dn_next_bonustype[i], ==, 0);
529         ASSERT3U(dn->dn_rm_spillblk[i], ==, 0);
530         ASSERT3U(dn->dn_next_blkisz[i], ==, 0);
531         ASSERT(!list_link_active(&dn->dn_dirty_link[i]));
532         ASSERT3P(list_head(&dn->dn_dirty_records[i]), ==, NULL);
533         ASSERT0(avl_numnodes(&dn->dn_ranges[i]));
534         ASSERT3U(avl_numnodes(&dn->dn_ranges[i]), ==, 0);
535     }
536
537     dn->dn_type = ot;
538     dnode_setdblks(dn, blocksize);
539     dn->dn_indblkshift = ibs;
540     dn->dn_nlevels = 1;
541     if (bonustype == DMU_OT_SA) /* Maximize bonus space for SA */
542         dn->dn_nblkptr = 1;
543     else
544         dn->dn_nblkptr = 1 +
545             ((DN_MAX_BONUSLEN - bonuslen) >> SPA_BLKPTRSHIFT);
546     dn->dn_bonustype = bonustype;
547     dn->dn_bonuslen = bonuslen;
548     dn->dn_checksum = ZIO_CHECKSUM_INHERIT;
549     dn->dn_compress = ZIO_COMPRESS_INHERIT;
550     dn->dn_dirtyctx = 0;
551
552     dn->dn_free_txg = 0;
553     if (dn->dn_dirtyctx_firstset) {
554         kmem_free(dn->dn_dirtyctx_firstset, 1);
555         dn->dn_dirtyctx_firstset = NULL;
556     }
557
558     dn->dn_allocated_txg = tx->tx_txg;
559     dn->dn_id_flags = 0;
560
561     dnode_setdirty(dn, tx);
562     dn->dn_next_indblkshift[tx->tx_txg & TXG_MASK] = ibs;
563     dn->dn_next_bonuslen[tx->tx_txg & TXG_MASK] = dn->dn_bonuslen;
564     dn->dn_next_bonustype[tx->tx_txg & TXG_MASK] = dn->dn_bonustype;
565     dn->dn_next_blkisz[tx->tx_txg & TXG_MASK] = dn->dn_datablks;
566 }
567
568 void
569 dnode_reallocate(dnode_t *dn, dmu_object_type_t ot, int blocksize,
570     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx)
571 {
572     int nblkptr;
573
574     ASSERT3U(blocksize, >=, SPA_MINBLOCKSIZE);
575     ASSERT3U(blocksize, <=, SPA_MAXBLOCKSIZE);
576     ASSERT0(blocksize % SPA_MINBLOCKSIZE);
577     ASSERT3U(blocksize % SPA_MINBLOCKSIZE, ==, 0);
578     ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT || dmu_tx_private_ok(tx));
579     ASSERT(tx->tx_txg != 0);
580     ASSERT((bonustype == DMU_OT_NONE && bonuslen == 0) ||
581         (bonustype != DMU_OT_NONE && bonuslen != 0) ||

```

```

573         (bonustype == DMU_OT_SA && bonuslen == 0));
574     ASSERT(DMU_OT_IS_VALID(bonustype));
575     ASSERT3U(bonuslen, <=, DN_MAX_BONUSLEN);
576
577     /* clean up any unreferenced dbufs */
578     dnode_evict_dbufs(dn);
579
580     dn->dn_id_flags = 0;
581
582     rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
583     dnode_setdirty(dn, tx);
584     if (dn->dn_datablks != blocksize) {
585         /* change blocksize */
586         ASSERT(dn->dn_maxblkid == 0 &&
587             (BP_IS_HOLE(&dn->dn_phys->dn_blkptr[0]) ||
588             dnode_block_freed(dn, 0)));
589         dnode_setdblks(dn, blocksize);
590         dn->dn_next_blkisz[tx->tx_txg&TXG_MASK] = blocksize;
591     }
592     if (dn->dn_bonuslen != bonuslen)
593         dn->dn_next_bonuslen[tx->tx_txg&TXG_MASK] = bonuslen;
594
595     if (bonustype == DMU_OT_SA) /* Maximize bonus space for SA */
596         nblkptr = 1;
597     else
598         nblkptr = 1 + ((DN_MAX_BONUSLEN - bonuslen) >> SPA_BLKPTRSHIFT);
599     if (dn->dn_bonustype != bonustype)
600         dn->dn_next_bonustype[tx->tx_txg&TXG_MASK] = bonustype;
601     if (dn->dn_nblkptr != nblkptr)
602         dn->dn_next_nblkptr[tx->tx_txg&TXG_MASK] = nblkptr;
603     if (dn->dn_phys->dn_flags & DNODE_FLAG_SPILL_BLKPTR) {
604         dbuf_rm_spill(dn, tx);
605         dnode_rm_spill(dn, tx);
606     }
607     rw_exit(&dn->dn_struct_rwlock);
608
609     /* change type */
610     dn->dn_type = ot;
611
612     /* change bonus size and type */
613     mutex_enter(&dn->dn_mtx);
614     dn->dn_bonustype = bonustype;
615     dn->dn_bonuslen = bonuslen;
616     dn->dn_nblkptr = nblkptr;
617     dn->dn_checksum = ZIO_CHECKSUM_INHERIT;
618     dn->dn_compress = ZIO_COMPRESS_INHERIT;
619     ASSERT3U(dn->dn_nblkptr, <=, DN_MAX_NBLKPTR);
620
621     /* fix up the bonus db_size */
622     if (dn->dn_bonus) {
623         dn->dn_bonus->db.db_size =
624             DN_MAX_BONUSLEN - (dn->dn_nblkptr-1) * sizeof(blkptr_t);
625         ASSERT(dn->dn_bonuslen <= dn->dn_bonus->db.db_size);
626     }
627
628     dn->dn_allocated_txg = tx->tx_txg;
629     mutex_exit(&dn->dn_mtx);
630 }
631
632 unchanged portion omitted
633
634 void
635 dnode_setdirty(dnode_t *dn, dmu_tx_t *tx)
636 {
637     objset_t *os = dn->dn_objset;
638     uint64_t txg = tx->tx_txg;

```

```

1207     if (DMU_OBJECT_IS_SPECIAL(dn->dn_object)) {
1208         dsl_dataset_dirty(os->os_dsl_dataset, tx);
1209         return;
1210     }

1212     DNODE_VERIFY(dn);

1214 #ifdef ZFS_DEBUG
1215     mutex_enter(&dn->dn_mtx);
1216     ASSERT(dn->dn_phys->dn_type == dn->dn_allocated_txg);
1217     ASSERT(dn->dn_free_txg == 0 || dn->dn_free_txg >= txg);
1218     mutex_exit(&dn->dn_mtx);
1219 #endif

1221     /*
1222     * Determine old uid/gid when necessary
1223     */
1224     dmu_objset_userquota_get_ids(dn, B_TRUE, tx);

1226     mutex_enter(&os->os_lock);

1228     /*
1229     * If we are already marked dirty, we're done.
1230     */
1231     if (list_link_active(&dn->dn_dirty_link[txg & TXG_MASK])) {
1232         mutex_exit(&os->os_lock);
1233         return;
1234     }

1236     ASSERT(!refcount_is_zero(&dn->dn_holds) || list_head(&dn->dn_dbufs));
1237     ASSERT(dn->dn_datablksize != 0);
1238     ASSERT0(dn->dn_next_bonuslen[txg&TXG_MASK]);
1239     ASSERT0(dn->dn_next_blksize[txg&TXG_MASK]);
1240     ASSERT0(dn->dn_next_bonustype[txg&TXG_MASK]);
1241     ASSERT3U(dn->dn_next_bonuslen[txg&TXG_MASK], ==, 0);
1242     ASSERT3U(dn->dn_next_blksize[txg&TXG_MASK], ==, 0);
1243     ASSERT3U(dn->dn_next_bonustype[txg&TXG_MASK], ==, 0);

1244     dprintf_ds(os->os_dsl_dataset, "obj=%llu txg=%llu\n",
1245         dn->dn_object, txg);

1246     if (dn->dn_free_txg > 0 && dn->dn_free_txg <= txg) {
1247         list_insert_tail(&os->os_free_dnodes[txg&TXG_MASK], dn);
1248     } else {
1249         list_insert_tail(&os->os_dirty_dnodes[txg&TXG_MASK], dn);
1250     }

1251     mutex_exit(&os->os_lock);

1253     /*
1254     * The dnode maintains a hold on its containing dbuf as
1255     * long as there are holds on it. Each instantiated child
1256     * dbuf maintains a hold on the dnode. When the last child
1257     * drops its hold, the dnode will drop its hold on the
1258     * containing dbuf. We add a "dirty hold" here so that the
1259     * dnode will hang around after we finish processing its
1260     * children.
1261     */
1262     VERIFY(dnode_add_ref(dn, (void *) (uintptr_t) tx->tx_txg));

1264     (void) dbuf_dirty(dn->dn_dbuf, tx);

1266     dsl_dataset_dirty(os->os_dsl_dataset, tx);
1267 }

```

unchanged portion omitted

```

1508 void
1509 dnode_free_range(dnode_t *dn, uint64_t off, uint64_t len, dmu_tx_t *tx)
1510 {
1511     dmu_buf_impl_t *db;
1512     uint64_t blkoff, blkid, nblks;
1513     int blksize, blkshift, head, tail;
1514     int trunc = FALSE;
1515     int epbs;

1517     rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1518     blksize = dn->dn_datablksize;
1519     blkshift = dn->dn_datablkshift;
1520     epbs = dn->dn_indblks - SPA_BLKPTRSHIFT;

1522     if (len == -1ULL) {
1523         len = UINTE64_MAX - off;
1524         trunc = TRUE;
1525     }

1527     /*
1528     * First, block align the region to free:
1529     */
1530     if (ISP2(blksize)) {
1531         head = P2NPHASE(off, blksize);
1532         blkoff = P2PHASE(off, blksize);
1533         if ((off >> blkshift) > dn->dn_maxblkid)
1534             goto out;
1535     } else {
1536         ASSERT(dn->dn_maxblkid == 0);
1537         if (off == 0 && len >= blksize) {
1538             /* Freeing the whole block; fast-track this request */
1539             blkid = 0;
1540             nblks = 1;
1541             goto done;
1542         } else if (off >= blksize) {
1543             /* Freeing past end-of-data */
1544             goto out;
1545         } else {
1546             /* Freeing part of the block. */
1547             head = blksize - off;
1548             ASSERT3U(head, >, 0);
1549         }
1550         blkoff = off;
1551     }
1552     /* zero out any partial block data at the start of the range */
1553     if (head) {
1554         ASSERT3U(blkoff + head, ==, blksize);
1555         if (len < head)
1556             head = len;
1557         if (dbuf_hold_impl(dn, 0, dbuf_whichblock(dn, off), TRUE,
1558             FTAG, &db) == 0) {
1559             caddr_t data;

1561             /* don't dirty if it isn't on disk and isn't dirty */
1562             if (db->db_last_dirty ||
1563                 (db->db_blkptr && !BP_IS_HOLE(db->db_blkptr))) {
1564                 rw_exit(&dn->dn_struct_rwlock);
1565                 dbuf_will_dirty(db, tx);
1566                 rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1567                 data = db->db_data;
1568                 bzero(data + blkoff, head);
1569             }
1570             dbuf_rele(db, FTAG);
1571         }
1572         off += head;
1573         len -= head;

```

```

1574     }
1575
1576     /* If the range was less than one block, we're done */
1577     if (len == 0)
1578         goto out;
1579
1580     /* If the remaining range is past end of file, we're done */
1581     if ((off >> blkshift) > dn->dn_maxblkid)
1582         goto out;
1583
1584     ASSERT(ISP2(blksz));
1585     if (trunc)
1586         tail = 0;
1587     else
1588         tail = P2PHASE(len, blksz);
1589
1590     ASSERT0(P2PHASE(off, blksz));
1591     ASSERT3U(P2PHASE(off, blksz), ==, 0);
1592     /* zero out any partial block data at the end of the range */
1593     if (tail) {
1594         if (len < tail)
1595             tail = len;
1596         if (dbuf_hold_impl(dn, 0, dbuf_whichblock(dn, off+len),
1597             TRUE, FTAG, &db) == 0) {
1598             /* don't dirty if not on disk and not dirty */
1599             if (db->db_last_dirty ||
1600                 (db->db_blkptr && !BP_IS_HOLE(db->db_blkptr))) {
1601                 rw_exit(&dn->dn_struct_rwlock);
1602                 dbuf_will_dirty(db, tx);
1603                 rw_enter(&dn->dn_struct_rwlock, RW_WRITER);
1604                 bzero(db->db_data, tail);
1605             }
1606             dbuf_rele(db, FTAG);
1607         }
1608         len -= tail;
1609     }
1610
1611     /* If the range did not include a full block, we are done */
1612     if (len == 0)
1613         goto out;
1614
1615     ASSERT(IS_P2ALIGNED(off, blksz));
1616     ASSERT(trunc || IS_P2ALIGNED(len, blksz));
1617     blkid = off >> blkshift;
1618     nblks = len >> blkshift;
1619     if (trunc)
1620         nblks += 1;
1621
1622     /*
1623     * Read in and mark all the level-1 indirects dirty,
1624     * so that they will stay in memory until syncing phase.
1625     * Always dirty the first and last indirect to make sure
1626     * we dirty all the partial indirects.
1627     */
1628     if (dn->dn_nlevels > 1) {
1629         uint64_t i, first, last;
1630         int shift = epbs + dn->dn_datablkshift;
1631
1632         first = blkid >> epbs;
1633         if (db = dbuf_hold_level(dn, 1, first, FTAG)) {
1634             dbuf_will_dirty(db, tx);
1635             dbuf_rele(db, FTAG);
1636         }
1637         if (trunc)
1638             last = dn->dn_maxblkid >> epbs;
1639     } else

```

```

1639         last = (blkid + nblks - 1) >> epbs;
1640         if (last > first && (db = dbuf_hold_level(dn, 1, last, FTAG))) {
1641             dbuf_will_dirty(db, tx);
1642             dbuf_rele(db, FTAG);
1643         }
1644         for (i = first + 1; i < last; i++) {
1645             uint64_t ibleft = i << shift;
1646             int err;
1647
1648             err = dnode_next_offset(dn,
1649                 DNODE_FIND_HAVELock, &ibleft, 1, 1, 0);
1650             i = ibleft >> shift;
1651             if (err == ESRCH || i >= last)
1652                 break;
1653             ASSERT(err == 0);
1654             db = dbuf_hold_level(dn, 1, i, FTAG);
1655             if (db) {
1656                 dbuf_will_dirty(db, tx);
1657                 dbuf_rele(db, FTAG);
1658             }
1659         }
1660     }
1661 done:
1662     /*
1663     * Add this range to the dnode range list.
1664     * We will finish up this free operation in the syncing phase.
1665     */
1666     mutex_enter(&dn->dn_mtx);
1667     dnode_clear_range(dn, blkid, nblks, tx);
1668     {
1669         free_range_t *rp, *found;
1670         avl_index_t where;
1671         avl_tree_t *tree = &dn->dn_ranges[tx->tx_txc&TXG_MASK];
1672
1673         /* Add new range to dn_ranges */
1674         rp = kmem_alloc(sizeof (free_range_t), KM_SLEEP);
1675         rp->fr_blkid = blkid;
1676         rp->fr_nblks = nblks;
1677         found = avl_find(tree, rp, &where);
1678         ASSERT(found == NULL);
1679         avl_insert(tree, rp, where);
1680         dprintf_dnode(dn, "blkid=%llu nblks=%llu txg=%llu\n",
1681             blkid, nblks, tx->tx_txc);
1682     }
1683     mutex_exit(&dn->dn_mtx);
1684
1685     dbuf_free_range(dn, blkid, blkid + nblks - 1, tx);
1686     dnode_setdirty(dn, tx);
1687 out:
1688     if (trunc && dn->dn_maxblkid >= (off >> blkshift))
1689         dn->dn_maxblkid = (off >> blkshift ? (off >> blkshift) - 1 : 0);
1690
1691     rw_exit(&dn->dn_struct_rwlock);
1692 }

```

unchanged\_portion\_omitted

```

1752 /* call from syncing context when we actually write/free space for this dnode */
1753 void
1754 dnode_diduse_space(dnode_t *dn, int64_t delta)
1755 {
1756     uint64_t space;
1757     dprintf_dnode(dn, "dn=%p dnp=%p used=%llu delta=%lld\n",
1758         dn, dn->dn_phys,
1759         (u_longlong_t)dn->dn_phys->dn_used,
1760         (longlong_t)delta);

```

```
1762     mutex_enter(&dn->dn_mtx);
1763     space = DN_USED_BYTES(dn->dn_phys);
1764     if (delta > 0) {
1765         ASSERT3U(space + delta, >=, space); /* no overflow */
1766     } else {
1767         ASSERT3U(space, >=, -delta); /* no underflow */
1768     }
1769     space += delta;
1770     if (spa_version(dn->dn_objset->os_spa) < SPA_VERSION_DNODE_BYTES) {
1771         ASSERT((dn->dn_phys->dn_flags & DNODE_FLAG_USED_BYTES) == 0);
1772         ASSERT0(P2PHASE(space, 1<<DEV_BSHIFT));
1773         ASSERT3U(P2PHASE(space, 1<<DEV_BSHIFT), ==, 0);
1774         dn->dn_phys->dn_used = space >> DEV_BSHIFT;
1775     } else {
1776         dn->dn_phys->dn_used = space;
1777         dn->dn_phys->dn_flags |= DNODE_FLAG_USED_BYTES;
1778     }
1779     mutex_exit(&dn->dn_mtx);
1780 }
1781
1782 _____  
1783 unchanged portion omitted
```

```

*****
19294 Thu Jul 26 14:19:12 2012
new/usr/src/uts/common/fs/zfs/dnode_sync.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
    unchanged_portion_omitted_
214 #endif

216 #define ALL -1

218 static int
219 free_children(dmu_buf_impl_t *db, uint64_t blkid, uint64_t nblks, int trunc,
220              dmu_tx_t *tx)
221 {
222     dnode_t *dn;
223     blkptr_t *bp;
224     dmu_buf_impl_t *subdb;
225     uint64_t start, end, dbstart, dbend, i;
226     int epbs, shift, err;
227     int all = TRUE;
228     int blocks_freed = 0;

230     /*
231     * There is a small possibility that this block will not be cached:
232     * 1 - if level > 1 and there are no children with level <= 1
233     * 2 - if we didn't get a dirty hold (because this block had just
234     *     finished being written -- and so had no holds), and then this
235     *     block got evicted before we got here.
236     */
237     if (db->db_state != DB_CACHED)
238         (void) dbuf_read(db, NULL, DB_RF_MUST_SUCCEED);

240     dbuf_release_bp(db);
241     bp = (blkptr_t *)db->db_data;

243     DB_DNODE_ENTER(db);
244     dn = DB_DNODE(db);
245     epbs = dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT;
246     shift = (db->db_level - 1) * epbs;
247     dbstart = db->db_blkid << epbs;
248     start = blkid >> shift;
249     if (dbstart < start) {
250         bp += start - dbstart;
251         all = FALSE;
252     } else {
253         start = dbstart;
254     }
255     dbend = ((db->db_blkid + 1) << epbs) - 1;
256     end = (blkid + nblks - 1) >> shift;
257     if (dbend <= end)
258         end = dbend;
259     else if (all)
260         all = trunc;
261     ASSERT3U(start, <=, end);

263     if (db->db_level == 1) {
264         FREE_VERIFY(db, start, end, tx);
265         blocks_freed = free_blocks(dn, bp, end-start+1, tx);
266         arc_buf_freeze(db->db_buf);
267         ASSERT(all || blocks_freed == 0 || db->db_last_dirty);
268         DB_DNODE_EXIT(db);
269         return (all ? ALL : blocks_freed);
270     }

272     for (i = start; i <= end; i++, bp++) {
273         if (BP_IS_HOLE(bp))

```

```

274         continue;
275         rw_enter(&dn->dn_struct_rwlock, RW_READER);
276         err = dbuf_hold_impl(dn, db->db_level-1, i, TRUE, FTAG, &subdb);
277         ASSERT0(err);
278         ASSERT3U(err, ==, 0);
279         rw_exit(&dn->dn_struct_rwlock);

280         if (free_children(subdb, blkid, nblks, trunc, tx) == ALL) {
281             ASSERT3P(subdb->db_blkptr, ==, bp);
282             blocks_freed += free_blocks(dn, bp, 1, tx);
283         } else {
284             all = FALSE;
285         }
286         dbuf_rele(subdb, FTAG);
287     }
288     DB_DNODE_EXIT(db);
289     arc_buf_freeze(db->db_buf);
290 #ifdef ZFS_DEBUG
291     bp -= (end-start)+1;
292     for (i = start; i <= end; i++, bp++) {
293         if (i == start && blkid != 0)
294             continue;
295         else if (i == end && !trunc)
296             continue;
297         ASSERT0(bp->blk_birth);
298         ASSERT3U(bp->blk_birth, ==, 0);
299     }
300 #endif
301     ASSERT(all || blocks_freed == 0 || db->db_last_dirty);
302     return (all ? ALL : blocks_freed);
303 }

304 /*
305 * free_range: Traverse the indicated range of the provided file
306 * and "free" all the blocks contained there.
307 */
308 static void
309 dnode_sync_free_range(dnode_t *dn, uint64_t blkid, uint64_t nblks, dmu_tx_t *tx)
310 {
311     blkptr_t *bp = dn->dn_phys->dn_blkptr;
312     dmu_buf_impl_t *db;
313     int trunc, start, end, shift, i, err;
314     int dnlevel = dn->dn_phys->dn_nlevels;

316     if (blkid > dn->dn_phys->dn_maxblkid)
317         return;

319     ASSERT(dn->dn_phys->dn_maxblkid < UINT64_MAX);
320     trunc = blkid + nblks > dn->dn_phys->dn_maxblkid;
321     if (trunc)
322         nblks = dn->dn_phys->dn_maxblkid - blkid + 1;

324     /* There are no indirect blocks in the object */
325     if (dnlevel == 1) {
326         if (blkid >= dn->dn_phys->dn_nblkptr) {
327             /* this range was never made persistent */
328             return;
329         }
330         ASSERT3U(blkid + nblks, <=, dn->dn_phys->dn_nblkptr);
331         (void) free_blocks(dn, bp + blkid, nblks, tx);
332         if (trunc) {
333             uint64_t off = (dn->dn_phys->dn_maxblkid + 1) *
334                 (dn->dn_phys->dn_datablkssize << SPA_MINBLOCKSHIFT);
335             dn->dn_phys->dn_maxblkid = (blkid ? blkid - 1 : 0);
336             ASSERT(off < dn->dn_phys->dn_maxblkid ||
337                 dn->dn_phys->dn_maxblkid == 0 ||

```

```

338         dnode_next_offset(dn, 0, &off, 1, 1, 0) != 0);
339     }
340     return;
341 }

343 shift = (dnlevel - 1) * (dn->dn_phys->dn_indblkshift - SPA_BLKPTRSHIFT);
344 start = blkid >> shift;
345 ASSERT(start < dn->dn_phys->dn_nblkptr);
346 end = (blkid + nblks - 1) >> shift;
347 bp += start;
348 for (i = start; i <= end; i++, bp++) {
349     if (BP_IS_HOLE(bp))
350         continue;
351     rw_enter(&dn->dn_struct_rwlock, RW_READER);
352     err = dbuf_hold_impl(dn, dnlevel-1, i, TRUE, FTAG, &db);
353     ASSERT0(err);
353     ASSERT3U(err, ==, 0);
354     rw_exit(&dn->dn_struct_rwlock);

356     if (free_children(db, blkid, nblks, trunc, tx) == ALL) {
357         ASSERT3P(db->db_blkptr, ==, bp);
358         (void) free_blocks(dn, bp, 1, tx);
359     }
360     dbuf_rele(db, FTAG);
361 }
362 if (trunc) {
363     uint64_t off = (dn->dn_phys->dn_maxblkid + 1) *
364         (dn->dn_phys->dn_datablkssize << SPA_MINBLOCKSHIFT);
365     dn->dn_phys->dn_maxblkid = (blkid ? blkid - 1 : 0);
366     ASSERT(off < dn->dn_phys->dn_maxblkid ||
367         dn->dn_phys->dn_maxblkid == 0 ||
368         dnode_next_offset(dn, 0, &off, 1, 1, 0) != 0);
369 }
370 }

```

unchanged portion omitted

```

463 static void
464 dnode_sync_free(dnode_t *dn, dmu_tx_t *tx)
465 {
466     int txgoff = tx->tx_txg & TXG_MASK;
467
468     ASSERT(dmu_tx_is_syncing(tx));
469
470     /*
471     * Our contents should have been freed in dnode_sync() by the
472     * free range record inserted by the caller of dnode_free().
473     */
474     ASSERT0(DN_USED_BYTES(dn->dn_phys));
474     ASSERT3U(DN_USED_BYTES(dn->dn_phys), ==, 0);
475     ASSERT(BP_IS_HOLE(dn->dn_phys->dn_blkptr));
476
477     dnode_undirty_dbufs(&dn->dn_dirty_records[txgoff]);
478     dnode_evict_dbufs(dn);
479     ASSERT3P(list_head(&dn->dn_dbufs), ==, NULL);
480
481     /*
482     * XXX - It would be nice to assert this, but we may still
483     * have residual holds from async evictions from the arc...
484     *
485     * zfs_obj_to_path() also depends on this being
486     * commented out.
487     *
488     * ASSERT3U(refcount_count(&dn->dn_holds), ==, 1);
489     */
490
491     /* Undirty next bits */

```

```

492     dn->dn_next_nlevels[txgoff] = 0;
493     dn->dn_next_indblkshift[txgoff] = 0;
494     dn->dn_next_blkisz[txgoff] = 0;
495
496     /* ASSERT(blkptrs are zero); */
497     ASSERT(dn->dn_phys->dn_type != DMU_OT_NONE);
498     ASSERT(dn->dn_type != DMU_OT_NONE);
499
500     ASSERT(dn->dn_free_txg > 0);
501     if (dn->dn_allocated_txg != dn->dn_free_txg)
502         dbuf_will_dirty(dn->dn_dbuf, tx);
503     bzero(dn->dn_phys, sizeof (dnode_phys_t));
504
505     mutex_enter(&dn->dn_mtx);
506     dn->dn_type = DMU_OT_NONE;
507     dn->dn_maxblkid = 0;
508     dn->dn_allocated_txg = 0;
509     dn->dn_free_txg = 0;
510     dn->dn_have_spill = B_FALSE;
511     mutex_exit(&dn->dn_mtx);
512
513     ASSERT(dn->dn_object != DMU_META_DNODE_OBJECT);
514
515     dnode_rele(dn, (void *) (uintptr_t) tx->tx_txg);
516     /*
517     * Now that we've released our hold, the dnode may
518     * be evicted, so we musn't access it.
519     */
520 }

```

unchanged portion omitted

```

*****
117870 Thu Jul 26 14:19:13 2012
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

779 uint64_t
780 dsl_dataset_create_sync_dd(dsl_dir_t *dd, dsl_dataset_t *origin,
781     uint64_t flags, dmu_tx_t *tx)
782 {
783     dsl_pool_t *dp = dd->dd_pool;
784     dmu_buf_t *dbuf;
785     dsl_dataset_phys_t *dsphys;
786     uint64_t dsobj;
787     objset_t *mos = dp->dp_meta_objset;

789     if (origin == NULL)
790         origin = dp->dp_origin_snap;

792     ASSERT(origin == NULL || origin->ds_dir->dd_pool == dp);
793     ASSERT(origin == NULL || origin->ds_phys->ds_num_children > 0);
794     ASSERT(dmu_tx_is_syncing(tx));
795     ASSERT(dd->dd_phys->dd_head_dataset_obj == 0);

797     dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
798         DMU_OT_DSL_DATASET, sizeof (dsl_dataset_phys_t), tx);
799     VERIFY(0 == dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
800     dmu_buf_will_dirty(dbuf, tx);
801     dsphys = dbuf->db_data;
802     bzero(dsphys, sizeof (dsl_dataset_phys_t));
803     dsphys->ds_dir_obj = dd->dd_object;
804     dsphys->ds_flags = flags;
805     dsphys->ds_fsid_guid = unique_create();
806     (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,
807         sizeof (dsphys->ds_guid));
808     dsphys->ds_snapnames_zapobj =
809         zap_create_norm(mos, U8_TEXTPREP_Toupper, DMU_OT_DSL_DS_SNAP_MAP,
810             DMU_OT_NONE, 0, tx);
811     dsphys->ds_creation_time = gethrestime_sec();
812     dsphys->ds_creation_txg = tx->tx_txg == TXG_INITIAL ? 1 : tx->tx_txg;

814     if (origin == NULL) {
815         dsphys->ds_deadlist_obj = dsl_deadlist_alloc(mos, tx);
816     } else {
817         dsl_dataset_t *ohds;

819         dsphys->ds_prev_snap_obj = origin->ds_object;
820         dsphys->ds_prev_snap_txg =
821             origin->ds_phys->ds_creation_txg;
822         dsphys->ds_referenced_bytes =
823             origin->ds_phys->ds_referenced_bytes;
824         dsphys->ds_compressed_bytes =
825             origin->ds_phys->ds_compressed_bytes;
826         dsphys->ds_uncompressed_bytes =
827             origin->ds_phys->ds_uncompressed_bytes;
828         dsphys->ds_bp = origin->ds_phys->ds_bp;
829         dsphys->ds_flags |= origin->ds_phys->ds_flags;

831         dmu_buf_will_dirty(origin->ds_dbuf, tx);
832         origin->ds_phys->ds_num_children++;

834         VERIFY0(dsl_dataset_hold_obj(dp,
834             VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
835                 origin->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ohds)));
836         dsphys->ds_deadlist_obj = dsl_deadlist_clone(&ohds->ds_deadlist,

```

```

837         dsphys->ds_prev_snap_txg, dsphys->ds_prev_snap_obj, tx);
838     dsl_dataset_rele(ohds, FTAG);

840     if (spa_version(dp->dp_spa) >= SPA_VERSION_NEXT_CLONES) {
841         if (origin->ds_phys->ds_next_clones_obj == 0) {
842             origin->ds_phys->ds_next_clones_obj =
843                 zap_create(mos,
844                     DMU_OT_NEXT_CLONES, DMU_OT_NONE, 0, tx);
845         }
846         VERIFY(0 == zap_add_int(mos,
847             origin->ds_phys->ds_next_clones_obj,
848             dsobj, tx));
849     }

851     dmu_buf_will_dirty(dd->dd_dbuf, tx);
852     dd->dd_phys->dd_origin_obj = origin->ds_object;
853     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
854         if (origin->ds_dir->dd_phys->dd_clones == 0) {
855             dmu_buf_will_dirty(origin->ds_dir->dd_dbuf, tx);
856             origin->ds_dir->dd_phys->dd_clones =
857                 zap_create(mos,
858                     DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
859         }
860         VERIFY0(zap_add_int(mos,
860             VERIFY3U(0, ==, zap_add_int(mos,
861                 origin->ds_dir->dd_phys->dd_clones, dsobj, tx)));
862     }
863 }

865     if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
866         dsphys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;

868     dmu_buf_rele(dbuf, FTAG);

870     dmu_buf_will_dirty(dd->dd_dbuf, tx);
871     dd->dd_phys->dd_head_dataset_obj = dsobj;

873     return (dsobj);
874 }

876 uint64_t
877 dsl_dataset_create_sync(dsl_dir_t *pdd, const char *lastname,
878     dsl_dataset_t *origin, uint64_t flags, cred_t *cr, dmu_tx_t *tx)
879 {
880     dsl_pool_t *dp = pdd->dd_pool;
881     uint64_t dsobj, ddoobj;
882     dsl_dir_t *dd;

884     ASSERT(lastname[0] != '@');

886     ddoobj = dsl_dir_create_sync(dp, pdd, lastname, tx);
887     VERIFY(0 == dsl_dir_open_obj(dp, ddoobj, lastname, FTAG, &dd));

889     dsobj = dsl_dataset_create_sync_dd(dd, origin, flags, tx);

891     dsl_deleg_set_create_perms(dd, tx, cr);

893     dsl_dir_close(dd, FTAG);

895     /*
896     * If we are creating a clone, make sure we zero out any stale
897     * data from the origin snapshots zil header.
898     */
899     if (origin != NULL) {
900         dsl_dataset_t *ds;
901         objset_t *os;

```

```

903     VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
904     VERIFY0(dmu_objset_from_ds(ds, &os));
903     VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
904     VERIFY3U(0, ==, dmu_objset_from_ds(ds, &os));
905     bzero(&os->os_zil_header, sizeof(os->os_zil_header));
906     dsl_dataset_dirty(ds, tx);
907     dsl_dataset_rele(ds, FTAG);
908 }

910     return(dsobj);
911 }

unchanged_portion_omitted

1487 static void
1488 remove_from_next_clones(dsl_dataset_t *ds, uint64_t obj, dmu_tx_t *tx)
1489 {
1490     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
1491     uint64_t count;
1492     int err;

1494     ASSERT(ds->ds_phys->ds_num_children >= 2);
1495     err = zap_remove_int(mos, ds->ds_phys->ds_next_clones_obj, obj, tx);
1496     /*
1497      * The err should not be ENOENT, but a bug in a previous version
1498      * of the code could cause upgrade_clones_cb() to not set
1499      * ds_next_snap_obj when it should, leading to a missing entry.
1500      * If we knew that the pool was created after
1501      * SPA_VERSION_NEXT_CLONES, we could assert that it isn't
1502      * ENOENT. However, at least we can check that we don't have
1503      * too many entries in the next_clones_obj even after failing to
1504      * remove this one.
1505      */
1506     if (err != ENOENT) {
1507         VERIFY0(err);
1507         VERIFY3U(err, ==, 0);
1508     }
1509     ASSERT0(zap_count(mos, ds->ds_phys->ds_next_clones_obj, &count));
1509     ASSERT3U(0, ==, zap_count(mos, ds->ds_phys->ds_next_clones_obj,
1510         &count));
1510     ASSERT3U(count, <=, ds->ds_phys->ds_num_children - 2);
1511 }

1513 static void
1514 dsl_dataset_remove_clones_key(dsl_dataset_t *ds, uint64_t mintxg, dmu_tx_t *tx)
1515 {
1516     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
1517     zap_cursor_t zc;
1518     zap_attribute_t za;

1520     /*
1521      * If it is the old version, dd_clones doesn't exist so we can't
1522      * find the clones, but dsl_dataset_remove_key() is a no-op so it
1523      * doesn't matter.
1524      */
1525     if (ds->ds_dir->dd_phys->dd_clones == 0)
1526         return;

1528     for (zap_cursor_init(&zc, mos, ds->ds_dir->dd_phys->dd_clones);
1529          zap_cursor_retrieve(&zc, &za) == 0;
1530          zap_cursor_advance(&zc)) {
1531         dsl_dataset_t *clone;

1533         VERIFY0(dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
1534             VERIFY3U(0, ==, dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
1535                 za.za_first_integer, FTAG, &clone));

```

```

1535         if (clone->ds_dir->dd_origin_txg > mintxg) {
1536             dsl_deadlist_remove_key(&clone->ds_deadlist,
1537                 mintxg, tx);
1538             dsl_dataset_remove_clones_key(clone, mintxg, tx);
1539         }
1540         dsl_dataset_rele(clone, FTAG);
1541     }
1542     zap_cursor_fini(&zc);
1543 }

unchanged_portion_omitted

1576 static void
1577 process_old_deadlist(dsl_dataset_t *ds, dsl_dataset_t *ds_prev,
1578     dsl_dataset_t *ds_next, boolean_t after_branch_point, dmu_tx_t *tx)
1579 {
1580     struct process_old_arg poa = { 0 };
1581     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1582     objset_t *mos = dp->dp_meta_objset;

1584     ASSERT(ds->ds_deadlist.dl_oldfmt);
1585     ASSERT(ds_next->ds_deadlist.dl_oldfmt);

1587     poa.ds = ds;
1588     poa.ds_prev = ds_prev;
1589     poa.after_branch_point = after_branch_point;
1590     poa.pio = zio_root(dp->dp_spa, NULL, NULL, ZIO_FLAG_MUSTSUCCEED);
1591     VERIFY0(bpobj_iterate(&ds_next->ds_deadlist.dl_bpobj,
1592         VERIFY3U(0, ==, bpobj_iterate(&ds_next->ds_deadlist.dl_bpobj,
1593             process_old_cb, &poa, tx));
1594     VERIFY0(zio_wait(poa.pio));
1594     VERIFY3U(zio_wait(poa.pio), ==, 0);
1594     ASSERT3U(poa.used, ==, ds->ds_phys->ds_unique_bytes);

1596     /* change snapused */
1597     dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
1598         -poa.used, -poa.comp, -poa.uncomp, tx);

1600     /* swap next's deadlist to our deadlist */
1601     dsl_deadlist_close(&ds->ds_deadlist);
1602     dsl_deadlist_close(&ds_next->ds_deadlist);
1603     SWITCH64(ds_next->ds_phys->ds_deadlist_obj,
1604         ds->ds_phys->ds_deadlist_obj);
1605     dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
1606     dsl_deadlist_open(&ds_next->ds_deadlist, mos,
1607         ds_next->ds_phys->ds_deadlist_obj);
1608 }

1610 static int
1611 old_synchronous_dataset_destroy(dsl_dataset_t *ds, dmu_tx_t *tx)
1612 {
1613     int err;
1614     struct killarg ka;

1616     /*
1617      * Free everything that we point to (that's born after
1618      * the previous snapshot, if we are a clone)
1619      *
1620      * NB: this should be very quick, because we already
1621      * freed all the objects in open context.
1622      */
1623     ka.ds = ds;
1624     ka.tx = tx;
1625     err = traverse_dataset(ds,
1626         ds->ds_phys->ds_prev_snap_txg, TRAVERSE_POST,
1627         kill_blkptr, &ka);
1628     ASSERT0(err);

```



```

1629     ASSERT3U(err, ==, 0);
1629     ASSERT(!DS_UNIQUE_IS_ACCURATE(ds) || ds->ds_phys->ds_unique_bytes == 0);

1631     return (err);
1632 }

1634 void
1635 dsl_dataset_destroy_sync(void *arg1, void *tag, dmu_tx_t *tx)
1636 {
1637     struct dsl_ds_destroyarg *dsda = arg1;
1638     dsl_dataset_t *ds = dsda->ds;
1639     int err;
1640     int after_branch_point = FALSE;
1641     dsl_pool_t *dp = ds->ds_dir->dd_pool;
1642     objset_t *mos = dp->dp_meta_objset;
1643     dsl_dataset_t *ds_prev = NULL;
1644     boolean_t wont_destroy;
1645     uint64_t obj;

1647     wont_destroy = (dsda->defer &&
1648         (ds->ds_userrefs > 0 || ds->ds_phys->ds_num_children > 1));

1650     ASSERT(ds->ds_owner || wont_destroy);
1651     ASSERT(dsda->defer || ds->ds_phys->ds_num_children <= 1);
1652     ASSERT(ds->ds_prev == NULL ||
1653         ds->ds_prev->ds_phys->ds_next_snap_obj != ds->ds_object);
1654     ASSERT3U(ds->ds_phys->ds_bp.blk_birth, <=, tx->tx_tngx);

1656     if (wont_destroy) {
1657         ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
1658         dmu_buf_will_dirty(ds->ds_dbuf, tx);
1659         ds->ds_phys->ds_flags |= DS_FLAG_DEFER_DESTROY;
1660         spa_history_log_internal(ds, "defer_destroy", tx, "");
1661         return;
1662     }

1664     /* We need to log before removing it from the namespace. */
1665     spa_history_log_internal(ds, "destroy", tx, "");

1667     /* signal any waiters that this dataset is going away */
1668     mutex_enter(&ds->ds_lock);
1669     ds->ds_owner = dsl_reaper;
1670     cv_broadcast(&ds->ds_exclusive_cv);
1671     mutex_exit(&ds->ds_lock);

1673     /* Remove our reservation */
1674     if (ds->ds_reserved != 0) {
1675         dsl_prop_setarg_t psa;
1676         uint64_t value = 0;

1678         dsl_prop_setarg_init_uint64(&psa, "reservation",
1679             (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED),
1680             &value);
1681         psa.psa_effective_value = 0; /* predict default value */

1683         dsl_dataset_set_reservation_sync(ds, &psa, tx);
1684         ASSERT0(ds->ds_reserved);
1685         ASSERT3U(ds->ds_reserved, ==, 0);
1685     }

1687     ASSERT(RW_WRITE_HELD(&dp->dp_config_rwlock));

1689     dsl_scan_ds_destroyed(ds, tx);

1691     obj = ds->ds_object;

```

```

1693     if (ds->ds_phys->ds_prev_snap_obj != 0) {
1694         if (ds->ds_prev) {
1695             ds_prev = ds->ds_prev;
1696         } else {
1697             VERIFY(0 == dsl_dataset_hold_obj(dp,
1698                 ds->ds_phys->ds_prev_snap_obj, FTAG, &ds_prev));
1699         }
1700     }
1701     after_branch_point =
1702         (ds_prev->ds_phys->ds_next_snap_obj != obj);

1703     dmu_buf_will_dirty(ds_prev->ds_dbuf, tx);
1704     if (after_branch_point &&
1705         ds_prev->ds_phys->ds_next_clones_obj != 0) {
1706         remove_from_next_clones(ds_prev, obj, tx);
1707         if (ds->ds_phys->ds_next_snap_obj != 0) {
1708             VERIFY(0 == zap_add_int(mos,
1709                 ds_prev->ds_phys->ds_next_clones_obj,
1710                 ds->ds_phys->ds_next_snap_obj, tx));
1711         }
1712     }
1713     if (after_branch_point &&
1714         ds->ds_phys->ds_next_snap_obj == 0) {
1715         /* This clone is toast. */
1716         ASSERT(ds_prev->ds_phys->ds_num_children > 1);
1717         ds_prev->ds_phys->ds_num_children--;

1719         /*
1720          * If the clone's origin has no other clones, no
1721          * user holds, and has been marked for deferred
1722          * deletion, then we should have done the necessary
1723          * destroy setup for it.
1724          */
1725         if (ds_prev->ds_phys->ds_num_children == 1 &&
1726             ds_prev->ds_userrefs == 0 &&
1727             DS_IS_DEFER_DESTROY(ds_prev)) {
1728             ASSERT3P(dsda->rm_origin, !=, NULL);
1729         } else {
1730             ASSERT3P(dsda->rm_origin, ==, NULL);
1731         }
1732     } else if (!after_branch_point) {
1733         ds_prev->ds_phys->ds_next_snap_obj =
1734             ds->ds_phys->ds_next_snap_obj;
1735     }
1736 }

1738     if (dsl_dataset_is_snapshot(ds)) {
1739         dsl_dataset_t *ds_next;
1740         uint64_t old_unique;
1741         uint64_t used = 0, comp = 0, uncomp = 0;

1743         VERIFY(0 == dsl_dataset_hold_obj(dp,
1744             ds->ds_phys->ds_next_snap_obj, FTAG, &ds_next));
1745         ASSERT3U(ds_next->ds_phys->ds_prev_snap_obj, ==, obj);

1747         old_unique = ds_next->ds_phys->ds_unique_bytes;

1749         dmu_buf_will_dirty(ds_next->ds_dbuf, tx);
1750         ds_next->ds_phys->ds_prev_snap_obj =
1751             ds->ds_phys->ds_prev_snap_obj;
1752         ds_next->ds_phys->ds_prev_snap_txg =
1753             ds->ds_phys->ds_prev_snap_txg;
1754         ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
1755             ds_prev ? ds_prev->ds_phys->ds_creation_txg : 0);

1758         if (ds_next->ds_deadlist.dl_oldfmt) {

```

```

1759     process_old_deadlist(ds, ds_prev, ds_next,
1760         after_branch_point, tx);
1761     } else {
1762         /* Adjust prev's unique space. */
1763         if (ds_prev && !after_branch_point) {
1764             dsl_deadlist_space_range(&ds_next->ds_deadlist,
1765                 ds_prev->ds_phys->ds_prev_snap_txg,
1766                 ds->ds_phys->ds_prev_snap_txg,
1767                 &used, &comp, &uncomp);
1768             ds_prev->ds_phys->ds_unique_bytes += used;
1769         }
1771         /* Adjust snapused. */
1772         dsl_deadlist_space_range(&ds_next->ds_deadlist,
1773             ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,
1774             &used, &comp, &uncomp);
1775         dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
1776             -used, -comp, -uncomp, tx);
1778         /* Move blocks to be freed to pool's free list. */
1779         dsl_deadlist_move_bproj(&ds_next->ds_deadlist,
1780             &dp->dp_free_bproj, ds->ds_phys->ds_prev_snap_txg,
1781             tx);
1782         dsl_dir_diduse_space(tx->tx_pool->dp_free_dir,
1783             DD_USED_HEAD, used, comp, uncomp, tx);
1785         /* Merge our deadlist into next's and free it. */
1786         dsl_deadlist_merge(&ds_next->ds_deadlist,
1787             ds->ds_phys->ds_deadlist_obj, tx);
1788     }
1789     dsl_deadlist_close(&ds->ds_deadlist);
1790     dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
1792     /* Collapse range in clone heads */
1793     dsl_dataset_remove_clones_key(ds,
1794         ds->ds_phys->ds_creation_txg, tx);
1796     if (dsl_dataset_is_snapshot(ds_next)) {
1797         dsl_dataset_t *ds_nextnext;
1799         /*
1800          * Update next's unique to include blocks which
1801          * were previously shared by only this snapshot
1802          * and it. Those blocks will be born after the
1803          * prev snap and before this snap, and will have
1804          * died after the next snap and before the one
1805          * after that (ie. be on the snap after next's
1806          * deadlist).
1807          */
1808         VERIFY0(0 == dsl_dataset_hold_obj(dp,
1809             ds_next->ds_phys->ds_next_snap_obj,
1810             FTAG, &ds_nextnext));
1811         dsl_deadlist_space_range(&ds_nextnext->ds_deadlist,
1812             ds->ds_phys->ds_prev_snap_txg,
1813             ds->ds_phys->ds_creation_txg,
1814             &used, &comp, &uncomp);
1815         ds_next->ds_phys->ds_unique_bytes += used;
1816         dsl_dataset_rele(ds_nextnext, FTAG);
1817         ASSERT3P(ds_next->ds_prev, ==, NULL);
1819         /* Collapse range in this head. */
1820         dsl_dataset_t *hds;
1821         VERIFY0(dsl_dataset_hold_obj(dp,
1822             VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
1823                 ds->ds_dir->dd_phys->dd_head_dataset_obj,
1824                 FTAG, &hds)));

```

```

1824         dsl_deadlist_remove_key(&hds->ds_deadlist,
1825             ds->ds_phys->ds_creation_txg, tx);
1826         dsl_dataset_rele(hds, FTAG);
1828     } else {
1829         ASSERT3P(ds_next->ds_prev, ==, ds);
1830         dsl_dataset_drop_ref(ds_next->ds_prev, ds_next);
1831         ds_next->ds_prev = NULL;
1832         if (ds_prev) {
1833             VERIFY(0 == dsl_dataset_get_ref(dp,
1834                 ds->ds_phys->ds_prev_snap_obj,
1835                 ds_next, &ds_next->ds_prev));
1836         }
1838         dsl_dataset_recalc_head_uniq(ds_next);
1840         /*
1841          * Reduce the amount of our unconsmred reservation
1842          * being charged to our parent by the amount of
1843          * new unique data we have gained.
1844          */
1845         if (old_unique < ds_next->ds_reserved) {
1846             int64_t mrsdelta;
1847             uint64_t new_unique =
1848                 ds_next->ds_phys->ds_unique_bytes;
1850             ASSERT(old_unique <= new_unique);
1851             mrsdelta = MIN(new_unique - old_unique,
1852                 ds_next->ds_reserved - old_unique);
1853             dsl_dir_diduse_space(ds->ds_dir,
1854                 DD_USED_REFRSRV, -mrsdelta, 0, 0, tx);
1855         }
1856     }
1857     dsl_dataset_rele(ds_next, FTAG);
1858 } else {
1859     zfeature_info_t *async_destroy =
1860         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY];
1862     /*
1863      * There's no next snapshot, so this is a head dataset.
1864      * Destroy the deadlist. Unless it's a clone, the
1865      * deadlist should be empty. (If it's a clone, it's
1866      * safe to ignore the deadlist contents.)
1867      */
1868     dsl_deadlist_close(&ds->ds_deadlist);
1869     dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
1870     ds->ds_phys->ds_deadlist_obj = 0;
1872     if (!spa_feature_is_enabled(dp->dp_spa, async_destroy)) {
1873         err = old_synchronous_dataset_destroy(ds, tx);
1874     } else {
1875         /*
1876          * Move the bptree into the pool's list of trees to
1877          * clean up and update space accounting information.
1878          */
1879         uint64_t used, comp, uncomp;
1881         ASSERT(err == 0 || err == EBUSY);
1882         if (!spa_feature_is_active(dp->dp_spa, async_destroy)) {
1883             spa_feature_incr(dp->dp_spa, async_destroy, tx);
1884             dp->dp_bptree_obj = bptree_alloc(
1885                 dp->dp_meta_objset, tx);
1886             VERIFY(zap_add(dp->dp_meta_objset,
1887                 DMU_POOL_DIRECTORY_OBJECT,
1888                 DMU_POOL_BPTREE_OBJ, sizeof(uint64_t), 1,
1889                 &dp->dp_bptree_obj, tx) == 0);

```

```

1890     }
1892     used = ds->ds_dir->dd_phys->dd_used_bytes;
1893     comp = ds->ds_dir->dd_phys->dd_compressed_bytes;
1894     uncomp = ds->ds_dir->dd_phys->dd_uncompressed_bytes;
1896     ASSERT(!DS_UNIQUE_IS_ACCURATE(ds) ||
1897           ds->ds_phys->ds_unique_bytes == used);
1899     bptree_add(dp->dp_meta_objset, dp->dp_bptree_obj,
1900             &ds->ds_phys->ds_bp, ds->ds_phys->ds_prev_snap_txg,
1901             used, comp, uncomp, tx);
1902     dsl_dir_diduse_space(ds->ds_dir, DD_USED_HEAD,
1903             -used, -comp, -uncomp, tx);
1904     dsl_dir_diduse_space(dp->dp_free_dir, DD_USED_HEAD,
1905             used, comp, uncomp, tx);
1906     }
1908     if (ds->ds_prev != NULL) {
1909         if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
1910             VERIFY0(zap_remove_int(mos,
1911                 VERIFY3U(0, ==, zap_remove_int(mos,
1912                     ds->ds_prev->ds_dir->dd_phys->dd_clones,
1913                     ds->ds_object, tx)));
1914             dsl_dataset_rele(ds->ds_prev, ds);
1915             ds->ds_prev = ds_prev = NULL;
1916         }
1917     }
1919     /*
1920     * This must be done after the dsl_traverse(), because it will
1921     * re-open the objset.
1922     */
1923     if (ds->ds_objset) {
1924         dmu_objset_evict(ds->ds_objset);
1925         ds->ds_objset = NULL;
1926     }
1928     if (ds->ds_dir->dd_phys->dd_head_dataset_obj == ds->ds_object) {
1929         /* Erase the link in the dir */
1930         dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
1931         ds->ds_dir->dd_phys->dd_head_dataset_obj = 0;
1932         ASSERT(ds->ds_phys->ds_snapnames_zapobj != 0);
1933         err = zap_destroy(mos, ds->ds_phys->ds_snapnames_zapobj, tx);
1934         ASSERT(err == 0);
1935     } else {
1936         /* remove from snapshot namespace */
1937         dsl_dataset_t *ds_head;
1938         ASSERT(ds->ds_phys->ds_snapnames_zapobj == 0);
1939         VERIFY0(ds == dsl_dataset_hold_obj(dp,
1940             ds->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ds_head));
1941         VERIFY0(ds == dsl_dataset_get_snapname(ds));
1942 #ifndef ZFS_DEBUG
1943         {
1944             uint64_t val;
1946             err = dsl_dataset_snap_lookup(ds_head,
1947                 ds->ds_snapname, &val);
1948             ASSERT0(err);
1949             ASSERT3U(err, ==, 0);
1950             ASSERT3U(val, ==, obj);
1951         }
1952 #endif
1953         err = dsl_dataset_snap_remove(ds_head, ds->ds_snapname, tx);
1954         ASSERT(err == 0);

```

```

1954         dsl_dataset_rele(ds_head, FTAG);
1955     }
1957     if (ds_prev && ds->ds_prev != ds_prev)
1958         dsl_dataset_rele(ds_prev, FTAG);
1960     spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);
1962     if (ds->ds_phys->ds_next_clones_obj != 0) {
1963         uint64_t count;
1964         ASSERT(0 == zap_count(mos,
1965             ds->ds_phys->ds_next_clones_obj, &count) && count == 0);
1966         VERIFY0(0 == dmu_object_free(mos,
1967             ds->ds_phys->ds_next_clones_obj, tx));
1968     }
1969     if (ds->ds_phys->ds_props_obj != 0)
1970         VERIFY0(0 == zap_destroy(mos, ds->ds_phys->ds_props_obj, tx));
1971     if (ds->ds_phys->ds_userrefs_obj != 0)
1972         VERIFY0(0 == zap_destroy(mos, ds->ds_phys->ds_userrefs_obj, tx));
1973     dsl_dir_close(ds->ds_dir, ds);
1974     ds->ds_dir = NULL;
1975     dsl_dataset_drain_refs(ds, tag);
1976     VERIFY0(0 == dmu_object_free(mos, obj, tx));
1978     if (dsda->rm_origin) {
1979         /*
1980         * Remove the origin of the clone we just destroyed.
1981         */
1982         struct dsl_ds_destroyarg ndsda = {0};
1984         ndsda.ds = dsda->rm_origin;
1985         dsl_dataset_destroy_sync(&ndsda, tag, tx);
1986     }
1987 }
1988 }
1989 }
1990 }
1991 }
1992 }
1993 }
1994 }
1995 }
1996 }
1997 }
1998 }
1999 }
2000 }
2001 }
2002 }
2003 }
2004 }
2005 }
2006 }
2007 }
2008 }
2009 }
2010 }
2011 }
2012 }
2013 }
2014 }
2015 }
2016 }
2017 }
2018 }
2019 }
2020 }
2021 }
2022 }
2023 }
2024 }
2025 }
2026 }
2027 }
2028 }
2029 }
2030 }
2031 }
2032 }
2033 }
2034 }
2035 }
2036 }
2037 }
2038 }
2039 }
2040 }
2041 }
2042 }
2043 }
2044 }
2045 }
2046 }
2047 }
2048 }
2049 }
2050 }
2051 }
2052 }
2053 }
2054 }
2055 void
2056 dsl_dataset_snapshot_sync(dsl_dataset_t *ds, const char *snapname,
2057     dmu_tx_t *tx)
2058 {
2059     dsl_pool_t *dp = ds->ds_dir->dd_pool;
2060     dmu_buf_t *dbuf;
2061     dsl_dataset_phys_t *dsphys;
2062     uint64_t dsobj, crtctxg;
2063     objset_t *mos = dp->dp_meta_objset;
2064     int err;
2066     ASSERT(RW_WRITE_HELD(&dp->dp_config_rwlock));
2068     /*
2069     * The origin's ds_creation_txg has to be < TXG_INITIAL
2070     */
2071     if (strcmp(snapname, ORIGIN_DIR_NAME) == 0)
2072         crtctxg = 1;
2073     else
2074         crtctxg = tx->tx_txg;
2076     dsobj = dmu_object_alloc(mos, DMU_OT_DSL_DATASET, 0,
2077         DMU_OT_DSL_DATASET, sizeof(dsl_dataset_phys_t), tx);
2078     VERIFY0(0 == dmu_bonus_hold(mos, dsobj, FTAG, &dbuf));
2079     dmu_buf_will_dirty(dbuf, tx);
2080     dsphys = dbuf->db_data;
2081     bzero(dsphys, sizeof(dsl_dataset_phys_t));
2082     dsphys->ds_dir_obj = ds->ds_dir->dd_object;
2083     dsphys->ds_fsid_guid = unique_create();
2084     (void) random_get_pseudo_bytes((void*)&dsphys->ds_guid,

```

unchanged portion omitted

```

2085     sizeof(dsphys->ds_guid));
2086 dsphys->ds_prev_snap_obj = ds->ds_phys->ds_prev_snap_obj;
2087 dsphys->ds_prev_snap_txg = ds->ds_phys->ds_prev_snap_txg;
2088 dsphys->ds_next_snap_obj = ds->ds_object;
2089 dsphys->ds_num_children = 1;
2090 dsphys->ds_creation_time = gethrestime_sec();
2091 dsphys->ds_creation_txg = crtxg;
2092 dsphys->ds_deadlist_obj = ds->ds_phys->ds_deadlist_obj;
2093 dsphys->ds_referenced_bytes = ds->ds_phys->ds_referenced_bytes;
2094 dsphys->ds_compressed_bytes = ds->ds_phys->ds_compressed_bytes;
2095 dsphys->ds_uncompressed_bytes = ds->ds_phys->ds_uncompressed_bytes;
2096 dsphys->ds_flags = ds->ds_phys->ds_flags;
2097 dsphys->ds_bp = ds->ds_phys->ds_bp;
2098 dmubuf_rele(dbuf, FTAG);

2100 ASSERT3U(ds->ds_prev != 0, ==, ds->ds_phys->ds_prev_snap_obj != 0);
2101 if (ds->ds_prev) {
2102     uint64_t next_clones_obj =
2103         ds->ds_prev->ds_phys->ds_next_clones_obj;
2104     ASSERT(ds->ds_prev->ds_phys->ds_next_snap_obj ==
2105         ds->ds_object ||
2106         ds->ds_prev->ds_phys->ds_num_children > 1);
2107     if (ds->ds_prev->ds_phys->ds_next_snap_obj == ds->ds_object) {
2108         dmubuf_will_dirty(ds->ds_prev->ds_dbuf, tx);
2109         ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
2110             ds->ds_prev->ds_phys->ds_creation_txg);
2111         ds->ds_prev->ds_phys->ds_next_snap_obj = dsobj;
2112     } else if (next_clones_obj != 0) {
2113         remove_from_next_clones(ds->ds_prev,
2114             dsphys->ds_next_snap_obj, tx);
2115         VERIFY0(zap_add_int(mos,
2116             VERIFY3U(0, ==, zap_add_int(mos,
2117                 next_clones_obj, dsobj, tx)));
2118     }
2119 }
2120 /*
2121  * If we have a reference-reservation on this dataset, we will
2122  * need to increase the amount of reservation being charged
2123  * since our unique space is going to zero.
2124  */
2125 if (ds->ds_reserved) {
2126     int64_t delta;
2127     ASSERT(DS_UNIQUE_IS_ACCURATE(ds));
2128     delta = MIN(ds->ds_phys->ds_unique_bytes, ds->ds_reserved);
2129     dsl_dir_diduse_space(ds->ds_dir, DD_USED_REFRSRV,
2130         delta, 0, 0, tx);
2131 }

2133 dmubuf_will_dirty(ds->ds_dbuf, tx);
2134 zfs_dbgmsg("taking snapshot %s@%s%llu; newkey=%llu",
2135     ds->ds_dir->dd_myname, snapname, dsobj,
2136     ds->ds_phys->ds_prev_snap_txg);
2137 ds->ds_phys->ds_deadlist_obj = dsl_deadlist_clone(&ds->ds_deadlist,
2138     UINTE64_MAX, ds->ds_phys->ds_prev_snap_obj, tx);
2139 dsl_deadlist_close(&ds->ds_deadlist);
2140 dsl_deadlist_open(&ds->ds_deadlist, mos, ds->ds_phys->ds_deadlist_obj);
2141 dsl_deadlist_add_key(&ds->ds_deadlist,
2142     ds->ds_phys->ds_prev_snap_txg, tx);

2144 ASSERT3U(ds->ds_phys->ds_prev_snap_txg, <, tx->tx_txg);
2145 ds->ds_phys->ds_prev_snap_obj = dsobj;
2146 ds->ds_phys->ds_prev_snap_txg = crtxg;
2147 ds->ds_phys->ds_unique_bytes = 0;
2148 if (spa_version(dp->dp_spa) >= SPA_VERSION_UNIQUE_ACCURATE)
2149     ds->ds_phys->ds_flags |= DS_FLAG_UNIQUE_ACCURATE;

```

```

2151     err = zap_add(mos, ds->ds_phys->ds_snapnames_zapobj,
2152         snapname, 8, 1, &dsobj, tx);
2153     ASSERT(err == 0);

2155     if (ds->ds_prev)
2156         dsl_dataset_drop_ref(ds->ds_prev, ds);
2157     VERIFY(0 == dsl_dataset_get_ref(dp,
2158         ds->ds_phys->ds_prev_snap_obj, ds, &ds->ds_prev));

2160     dsl_scan_ds_snapshotted(ds, tx);

2162     dsl_dir_snap_cmtime_update(ds->ds_dir);

2164     spa_history_log_internal_ds(ds->ds_prev, "snapshot", tx, "");
2165 }
2166 unchanged portion omitted

2185 static void
2186 get_clones_stat(dsl_dataset_t *ds, nvlist_t *nv)
2187 {
2188     uint64_t count = 0;
2189     objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
2190     zap_cursor_t zc;
2191     zap_attribute_t za;
2192     nvlist_t *propval;
2193     nvlist_t *val;

2195     rw_enter(&ds->ds_dir->dd_pool->dp_config_rwlock, RW_READER);
2196     VERIFY(nvlist_alloc(&propval, NV_UNIQUE_NAME, KM_SLEEP) == 0);
2197     VERIFY(nvlist_alloc(&val, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2199     /*
2200      * There may be missing entries in ds_next_clones_obj
2201      * due to a bug in a previous version of the code.
2202      * Only trust it if it has the right number of entries.
2203      */
2204     if (ds->ds_phys->ds_next_clones_obj != 0) {
2205         ASSERT0(zap_count(mos, ds->ds_phys->ds_next_clones_obj,
2206             ASSERT3U(0, ==, zap_count(mos, ds->ds_phys->ds_next_clones_obj,
2207                 &count)));
2208     }
2209     if (count != ds->ds_phys->ds_num_children - 1) {
2210         goto fail;
2211     }
2212     for (zap_cursor_init(&zc, mos, ds->ds_phys->ds_next_clones_obj);
2213         zap_cursor_retrieve(&zc, &za) == 0;
2214         zap_cursor_advance(&zc)) {
2215         dsl_dataset_t *clone;
2216         char buf[ZFS_MAXNAMELEN];
2217         /*
2218          * Even though we hold the dp_config_rwlock, the dataset
2219          * may fail to open, returning ENOENT. If there is a
2220          * thread concurrently attempting to destroy this
2221          * dataset, it will have the ds_rwlock held for
2222          * RW_WRITER. Our call to dsl_dataset_hold_obj() ->
2223          * dsl_dataset_hold_ref() will fail its
2224          * rw_tryenter(&ds->ds_rwlock, RW_READER), drop the
2225          * dp_config_rwlock, and wait for the destroy progress
2226          * and signal ds_exclusive_cv. If the destroy was
2227          * successful, we will see that
2228          * DSL_DATASET_IS_DESTROYED(), and return ENOENT.
2229          */
2230         if (dsl_dataset_hold_obj(ds->ds_dir->dd_pool,
2231             za.za_first_integer, FTAG, &clone) != 0)
2232             continue;

```

```

2232         dsl_dir_name(clone->ds_dir, buf);
2233         VERIFY(nvlist_add_boolean(val, buf) == 0);
2234         dsl_dataset_rele(clone, FTAG);
2235     }
2236     zap_cursor_fini(&zsc);
2237     VERIFY(nvlist_add_nvlist(propval, ZPROP_VALUE, val) == 0);
2238     VERIFY(nvlist_add_nvlist(nv, zfs_prop_to_name(ZFS_PROP_CLONES),
2239         propval) == 0);
2240 fail:
2241     nvlist_free(val);
2242     nvlist_free(propval);
2243     rw_exit(&ds->ds_dir->dd_pool->dp_config_rwlock);
2244 }
    unchanged portion omitted

2427 static void
2428 dsl_dataset_snapshot_rename_sync(void *arg1, void *arg2, dmu_tx_t *tx)
2429 {
2430     dsl_dataset_t *ds = arg1;
2431     const char *newsnapname = arg2;
2432     dsl_dir_t *dd = ds->ds_dir;
2433     objset_t *mos = dd->dd_pool->dp_meta_objset;
2434     dsl_dataset_t *hds;
2435     int err;

2437     ASSERT(ds->ds_phys->ds_next_snap_obj != 0);

2439     VERIFY(0 == dsl_dataset_hold_obj(dd->dd_pool,
2440         dd->dd_phys->dd_head_dataset_obj, FTAG, &hds));

2442     VERIFY(0 == dsl_dataset_get_snapname(ds));
2443     err = dsl_dataset_snap_remove(hds, ds->ds_snapname, tx);
2444     ASSERT0(err);
2445     ASSERT3U(err, ==, 0);
2446     mutex_enter(&ds->ds_lock);
2447     (void) strcpy(ds->ds_snapname, newsnapname);
2448     mutex_exit(&ds->ds_lock);
2449     err = zap_add(mos, hds->ds_phys->ds_snapnames_zapobj,
2450         ds->ds_snapname, 8, 1, &ds->ds_object, tx);
2451     ASSERT0(err);
2452     ASSERT3U(err, ==, 0);

2452     spa_history_log_internal(ds, "rename", tx,
2453         "-> %s", newsnapname);
2454     dsl_dataset_rele(hds, FTAG);
2455 }
    unchanged portion omitted

2772 static void
2773 dsl_dataset_promote_sync(void *arg1, void *arg2, dmu_tx_t *tx)
2774 {
2775     dsl_dataset_t *hds = arg1;
2776     struct promotearg *pa = arg2;
2777     struct promotenode *snap = list_head(&pa->shared_snaps);
2778     dsl_dataset_t *origin_ds = snap->ds;
2779     dsl_dataset_t *origin_head;
2780     dsl_dir_t *dd = hds->ds_dir;
2781     dsl_pool_t *dp = hds->ds_dir->dd_pool;
2782     dsl_dir_t *odd = NULL;
2783     uint64_t oldnext_obj;
2784     int64_t delta;

2786     ASSERT(0 == (hds->ds_phys->ds_flags & DS_FLAG_NOPROMOTE));

2788     snap = list_head(&pa->origin_snaps);
2789     origin_head = snap->ds;

```

```

2791     /*
2792     * We need to explicitly open odd, since origin_ds's dd will be
2793     * changing.
2794     */
2795     VERIFY(0 == dsl_dir_open_obj(dp, origin_ds->ds_dir->dd_object,
2796         NULL, FTAG, &odd));

2798     /* change origin's next snap */
2799     dmu_buf_will_dirty(origin_ds->ds_dbuf, tx);
2800     oldnext_obj = origin_ds->ds_phys->ds_next_snap_obj;
2801     snap = list_tail(&pa->clone_snaps);
2802     ASSERT3U(snap->ds->ds_phys->ds_prev_snap_obj, ==, origin_ds->ds_object);
2803     origin_ds->ds_phys->ds_next_snap_obj = snap->ds->ds_object;

2805     /* change the origin's next clone */
2806     if (origin_ds->ds_phys->ds_next_clones_obj) {
2807         remove_from_next_clones(origin_ds, snap->ds->ds_object, tx);
2808         VERIFY0(zap_add_int(dp->dp_meta_objset,
2809             VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2810                 origin_ds->ds_phys->ds_next_clones_obj,
2811                 oldnext_obj, tx));
2812         )
2813     }

2813     /* change origin */
2814     dmu_buf_will_dirty(dd->dd_dbuf, tx);
2815     ASSERT3U(dd->dd_phys->dd_origin_obj, ==, origin_ds->ds_object);
2816     dd->dd_phys->dd_origin_obj = odd->dd_phys->dd_origin_obj;
2817     dd->dd_origin_txg = origin_head->ds_dir->dd_origin_txg;
2818     dmu_buf_will_dirty(odd->dd_dbuf, tx);
2819     odd->dd_phys->dd_origin_obj = origin_ds->ds_object;
2820     origin_head->ds_dir->dd_origin_txg =
2821         origin_ds->ds_phys->ds_creation_txg;

2823     /* change dd_clone entries */
2824     if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2825         VERIFY0(zap_remove_int(dp->dp_meta_objset,
2826             VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
2827                 odd->dd_phys->dd_clones, hds->ds_object, tx));
2828         )
2829         VERIFY0(zap_add_int(dp->dp_meta_objset,
2830             VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2831                 pa->origin_origin->ds_dir->dd_phys->dd_clones,
2832                 hds->ds_object, tx));
2833         )
2834     }

2831     VERIFY0(zap_remove_int(dp->dp_meta_objset,
2832         VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
2833             pa->origin_origin->ds_dir->dd_phys->dd_clones,
2834             origin_head->ds_object, tx));
2835     if (dd->dd_phys->dd_clones == 0) {
2836         dd->dd_phys->dd_clones = zap_create(dp->dp_meta_objset,
2837             DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
2838     }
2839     VERIFY0(zap_add_int(dp->dp_meta_objset,
2840         VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
2841             dd->dd_phys->dd_clones, origin_head->ds_object, tx));
2842     )

2843     /* move snapshots to this dir */
2844     for (snap = list_head(&pa->shared_snaps); snap;
2845         snap = list_next(&pa->shared_snaps, snap)) {
2846         dsl_dataset_t *ds = snap->ds;

2848         /* unregister props as dsl_dir is changing */
2849         if (ds->ds_objset) {
2850             dmu_objset_evict(ds->ds_objset);

```

```

2851         ds->ds_objset = NULL;
2852     }
2853     /* move snap name entry */
2854     VERIFY(0 == dsl_dataset_get_snapname(ds));
2855     VERIFY(0 == dsl_dataset_snap_remove(origin_head,
2856         ds->ds_snapname, tx));
2857     VERIFY(0 == zap_add(dp->dp_meta_objset,
2858         hds->ds_phys->ds_snapnames_zapobj, ds->ds_snapname,
2859         8, 1, &ds->ds_object, tx));

2861     /* change containing dsl_dir */
2862     dmu_buf_will_dirty(ds->ds_dbuf, tx);
2863     ASSERT3U(ds->ds_phys->ds_dir_obj, ==, odd->dd_object);
2864     ds->ds_phys->ds_dir_obj = dd->dd_object;
2865     ASSERT3P(ds->ds_dir, ==, odd);
2866     dsl_dir_close(ds->ds_dir, ds);
2867     VERIFY(0 == dsl_dir_open_obj(dp, dd->dd_object,
2868         NULL, ds, &ds->ds_dir));

2870     /* move any clone references */
2871     if (ds->ds_phys->ds_next_clones_obj &&
2872         spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
2873         zap_cursor_t zc;
2874         zap_attribute_t za;

2876         for (zap_cursor_init(&zc, dp->dp_meta_objset,
2877             ds->ds_phys->ds_next_clones_obj);
2878             zap_cursor_retrieve(&zc, &za) == 0;
2879             zap_cursor_advance(&zc)) {
2880             dsl_dataset_t *cnds;
2881             uint64_t o;

2883             if (za.za_first_integer == oldnext_obj) {
2884                 /*
2885                  * We've already moved the
2886                  * origin's reference.
2887                  */
2888                 continue;
2889             }

2891             VERIFY0(dsl_dataset_hold_obj(dp,
2892                 VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
2893                     za.za_first_integer, FTAG, &cnds)));
2894             o = cnds->ds_dir->dd_phys->dd_head_dataset_obj;

2895             VERIFY3U(zap_remove_int(dp->dp_meta_objset,
2896                 odd->dd_phys->dd_clones, o, tx), ==, 0);
2897             VERIFY3U(zap_add_int(dp->dp_meta_objset,
2898                 dd->dd_phys->dd_clones, o, tx), ==, 0);
2899             dsl_dataset_rele(cnds, FTAG);
2900         }
2901         zap_cursor_fini(&zc);
2902     }

2904     ASSERT0(dsl_prop_numcb(ds));
2905     ASSERT3U(dsl_prop_numcb(ds), ==, 0);
2906 }

2907 /*
2908  * Change space accounting.
2909  * Note, pa->*usedsnap and dd_used_breakdown[SNAP] will either
2910  * both be valid, or both be 0 (resulting in delta == 0). This
2911  * is true for each of {clone,origin} independently.
2912  */

2914     delta = pa->cloneusedsnap -

```

```

2915         dd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2916     ASSERT3S(delta, >=, 0);
2917     ASSERT3U(pa->used, >=, delta);
2918     dsl_dir_diduse_space(dd, DD_USED_SNAP, delta, 0, 0, tx);
2919     dsl_dir_diduse_space(dd, DD_USED_HEAD,
2920         pa->used - delta, pa->comp, pa->uncomp, tx);

2922     delta = pa->originusedsnap -
2923         odd->dd_phys->dd_used_breakdown[DD_USED_SNAP];
2924     ASSERT3S(delta, <=, 0);
2925     ASSERT3U(pa->used, >=, -delta);
2926     dsl_dir_diduse_space(odd, DD_USED_SNAP, delta, 0, 0, tx);
2927     dsl_dir_diduse_space(odd, DD_USED_HEAD,
2928         -pa->used - delta, -pa->comp, -pa->uncomp, tx);

2930     origin_ds->ds_phys->ds_unique_bytes = pa->unique;

2932     /* log history record */
2933     spa_history_log_internal_ds(hds, "promote", tx, "");

2935     dsl_dir_close(odd, FTAG);
2936 }
_____ unchanged_portion_omitted _____

3599 void
3600 dsl_register_onexit_hold_cleanup(dsl_dataset_t *ds, const char *htag,
3601     minor_t minor)
3602 {
3603     zfs_hold_cleanup_arg_t *ca;

3605     ca = kmem_alloc(sizeof (zfs_hold_cleanup_arg_t), KM_SLEEP);
3606     ca->dp = ds->ds_dir->dd_pool;
3607     ca->dsobj = ds->ds_object;
3608     (void) strcpy(ca->htag, htag, sizeof (ca->htag));
3609     VERIFY0(zfs_onexit_add_cb(minor,
3610         VERIFY3U(0, ==, zfs_onexit_add_cb(minor,
3611             dsl_dataset_user_release_onexit, ca, NULL)));
3611 }
_____ unchanged_portion_omitted _____

```

```

*****
12683 Thu Jul 26 14:19:14 2012
new/usr/src/uts/common/fs/zfs/dsl_deadlist.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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) 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 * Copyright (c) 2011 by Delphix. All rights reserved.
25 */

26 #include <sys/dsl_dataset.h>
27 #include <sys/dmu.h>
28 #include <sys/refcount.h>
29 #include <sys/zap.h>
30 #include <sys/zfs_context.h>
31 #include <sys/dsl_pool.h>

33 /*
34  * Deadlist concurrency:
35  *
36  * Deadlists can only be modified from the syncing thread.
37  *
38  * Except for dsl_deadlist_insert(), it can only be modified with the
39  * dp_config_rwlock held with RW_WRITER.
40  *
41  * The accessors (dsl_deadlist_space() and dsl_deadlist_space_range()) can
42  * be called concurrently, from open context, with the dl_config_rwlock held
43  * with RW_READER.
44  *
45  * Therefore, we only need to provide locking between dsl_deadlist_insert() and
46  * the accessors, protecting:
47  *     dl_phys->dl_used,comp,uncomp
48  *     and protecting the dl_tree from being loaded.
49  * The locking is provided by dl_lock. Note that locking on the bpobj_t
50  * provides its own locking, and dl_oldfmt is immutable.
51  */

53 static int
54 dsl_deadlist_compare(const void *arg1, const void *arg2)
55 {
56     const dsl_deadlist_entry_t *dle1 = arg1;
57     const dsl_deadlist_entry_t *dle2 = arg2;

59     if (dle1->dle_mintxg < dle2->dle_mintxg)
60         return (-1);

```

```

61     else if (dle1->dle_mintxg > dle2->dle_mintxg)
62         return (+1);
63     else
64         return (0);
65 }

67 static void
68 dsl_deadlist_load_tree(dsl_deadlist_t *dl)
69 {
70     zap_cursor_t zc;
71     zap_attribute_t za;

73     ASSERT(!dl->dl_oldfmt);
74     if (dl->dl_havetree)
75         return;

77     avl_create(&dl->dl_tree, dsl_deadlist_compare,
78             sizeof(dsl_deadlist_entry_t),
79             offsetof(dsl_deadlist_entry_t, dle_node));
80     for (zap_cursor_init(&zc, dl->dl_os, dl->dl_object);
81         zap_cursor_retrieve(&zc, &za) == 0;
82         zap_cursor_advance(&zc)) {
83         dsl_deadlist_entry_t *dle = kmem_alloc(sizeof(*dle), KM_SLEEP);
84         dle->dle_mintxg = strtonum(za.za_name, NULL);
85         VERIFY0(bpobj_open(&dle->dle_bpobj, dl->dl_os,
86             VERIFY3U(0, ==, bpobj_open(&dle->dle_bpobj, dl->dl_os,
87                 za.za_first_integer)));
88         avl_add(&dl->dl_tree, dle);
89     }
90     zap_cursor_fini(&zc);
91     dl->dl_havetree = B_TRUE;

93 void
94 dsl_deadlist_open(dsl_deadlist_t *dl, objset_t *os, uint64_t object)
95 {
96     dmu_object_info_t doi;

98     mutex_init(&dl->dl_lock, NULL, MUTEX_DEFAULT, NULL);
99     dl->dl_os = os;
100    dl->dl_object = object;
101    VERIFY0(dmu_bonus_hold(os, object, dl, &dl->dl_dbuf));
102    VERIFY3U(0, ==, dmu_bonus_hold(os, object, dl, &dl->dl_dbuf));
103    dmu_object_info_from_db(dl->dl_dbuf, &doi);
104    if (doi.doi_type == DMU_OT_BPBJ) {
105        dmu_buf_rele(dl->dl_dbuf, dl);
106        dl->dl_dbuf = NULL;
107        dl->dl_oldfmt = B_TRUE;
108        VERIFY0(bpobj_open(&dl->dl_bpobj, os, object));
109        VERIFY3U(0, ==, bpobj_open(&dl->dl_bpobj, os, object));
110        return;
111    }
112    dl->dl_oldfmt = B_FALSE;
113    dl->dl_phys = dl->dl_dbuf->db_data;
114    dl->dl_havetree = B_FALSE;
115 }

unchanged_portion_omitted

151 void
152 dsl_deadlist_free(objset_t *os, uint64_t dlobj, dmu_tx_t *tx)
153 {
154     dmu_object_info_t doi;
155     zap_cursor_t zc;
156     zap_attribute_t za;

```

```

158 VERIFY0(dmu_object_info(os, dlobj, &doi));
158 VERIFY3U(0, ==, dmu_object_info(os, dlobj, &doi));
159 if (doi.doi_type == DMU_OT_BPOBJ) {
160     bplib_free(os, dlobj, tx);
161     return;
162 }

164 for (zap_cursor_init(&zsc, os, dlobj);
165     zap_cursor_retrieve(&zsc, &za) == 0;
166     zap_cursor_advance(&zsc))
167     bplib_free(os, za.za_first_integer, tx);
168 zap_cursor_fini(&zsc);
169 VERIFY0(dmu_object_free(os, dlobj, tx));
169 VERIFY3U(0, ==, dmu_object_free(os, dlobj, tx));
170 }
unchanged portion omitted

203 /*
204  * Insert new key in deadlist, which must be > all current entries.
205  * mintxg is not inclusive.
206  */
207 void
208 dsl_deadlist_add_key(dsl_deadlist_t *dl, uint64_t mintxg, dmu_tx_t *tx)
209 {
210     uint64_t obj;
211     dsl_deadlist_entry_t *dle;

213     if (dl->dl_oldfmt)
214         return;

216     dsl_deadlist_load_tree(dl);

218     dle = kmem_alloc(sizeof (*dle), KM_SLEEP);
219     dle->dle_mintxg = mintxg;
220     obj = bplib_alloc(dl->dl_os, SPA_MAXBLOCKSIZE, tx);
221     VERIFY0(bplib_open(&dle->dle_bplib, dl->dl_os, obj));
222     VERIFY3U(0, ==, bplib_open(&dle->dle_bplib, dl->dl_os, obj));
223     avl_add(&dl->dl_tree, dle);

224     VERIFY0(zap_add_int_key(dl->dl_os, dl->dl_object,
224     VERIFY3U(0, ==, zap_add_int_key(dl->dl_os, dl->dl_object,
225     mintxg, obj, tx));
226 }

228 /*
229  * Remove this key, merging its entries into the previous key.
230  */
231 void
232 dsl_deadlist_remove_key(dsl_deadlist_t *dl, uint64_t mintxg, dmu_tx_t *tx)
233 {
234     dsl_deadlist_entry_t dle_tofind;
235     dsl_deadlist_entry_t *dle, *dle_prev;

237     if (dl->dl_oldfmt)
238         return;

240     dsl_deadlist_load_tree(dl);

242     dle_tofind.dle_mintxg = mintxg;
243     dle = avl_find(&dl->dl_tree, &dle_tofind, NULL);
244     dle_prev = AVL_PREV(&dl->dl_tree, dle);

246     bplib_enqueue_subobj(&dle_prev->dle_bplib,
247     dle->dle_bplib.bpo_object, tx);

249     avl_remove(&dl->dl_tree, dle);

```

```

250     bplib_close(&dle->dle_bplib);
251     kmem_free(dle, sizeof (*dle));

253     VERIFY0(zap_remove_int(dl->dl_os, dl->dl_object, mintxg, tx));
253     VERIFY3U(0, ==, zap_remove_int(dl->dl_os, dl->dl_object, mintxg, tx));
254 }

256 /*
257  * Walk ds's snapshots to regenerate generate ZAP & AVL.
258  */
259 static void
260 dsl_deadlist_regenerate(objset_t *os, uint64_t dlobj,
261     uint64_t mrs_obj, dmu_tx_t *tx)
262 {
263     dsl_deadlist_t dl;
264     dsl_pool_t *dp = dmu_objset_pool(os);

266     dsl_deadlist_open(&dl, os, dlobj);
267     if (dl.dl_oldfmt) {
268         dsl_deadlist_close(&dl);
269         return;
270     }

272     while (mrs_obj != 0) {
273         dsl_dataset_t *ds;
274         VERIFY0(dsl_dataset_hold_obj(dp, mrs_obj, FTAG, &ds));
274         VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, mrs_obj, FTAG, &ds));
275         dsl_deadlist_add_key(&dl, ds->ds_phys->ds_prev_snap_txg, tx);
276         mrs_obj = ds->ds_phys->ds_prev_snap_obj;
277         dsl_dataset_rele(ds, FTAG);
278     }
279     dsl_deadlist_close(&dl);
280 }

282 uint64_t
283 dsl_deadlist_clone(dsl_deadlist_t *dl, uint64_t maxtxg,
284     uint64_t mrs_obj, dmu_tx_t *tx)
285 {
286     dsl_deadlist_entry_t *dle;
287     uint64_t newobj;

289     newobj = dsl_deadlist_alloc(dl->dl_os, tx);

291     if (dl->dl_oldfmt) {
292         dsl_deadlist_regenerate(dl->dl_os, newobj, mrs_obj, tx);
293         return (newobj);
294     }

296     dsl_deadlist_load_tree(dl);

298     for (dle = avl_first(&dl->dl_tree); dle;
299         dle = AVL_NEXT(&dl->dl_tree, dle)) {
300         uint64_t obj;

302         if (dle->dle_mintxg >= maxtxg)
303             break;

305         obj = bplib_alloc(dl->dl_os, SPA_MAXBLOCKSIZE, tx);
306         VERIFY0(zap_add_int_key(dl->dl_os, newobj,
306         VERIFY3U(0, ==, zap_add_int_key(dl->dl_os, newobj,
307         dle->dle_mintxg, obj, tx));
308     }
309     return (newobj);
310 }

312 void

```



```

313 dsl_deadlist_space(dsl_deadlist_t *dl,
314     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
315 {
316     if (dl->dl_oldfmt) {
317         VERIFY0(bpobj_space(&dl->dl_bpobj,
318             VERIFY3U(0, ==, bpobj_space(&dl->dl_bpobj,
319                 usedp, compp, uncompp)));
320     }
321
322     mutex_enter(&dl->dl_lock);
323     *usedp = dl->dl_phys->dl_used;
324     *compp = dl->dl_phys->dl_comp;
325     *uncompp = dl->dl_phys->dl_uncomp;
326     mutex_exit(&dl->dl_lock);
327 }
328
329 /*
330  * return space used in the range (mintxg, maxtxg].
331  * Includes maxtxg, does not include mintxg.
332  * mintxg and maxtxg must both be keys in the deadlist (unless maxtxg is
333  * larger than any bp in the deadlist (eg. UINTE64_MAX)).
334  */
335 void
336 dsl_deadlist_space_range(dsl_deadlist_t *dl, uint64_t mintxg, uint64_t maxtxg,
337     uint64_t *usedp, uint64_t *compp, uint64_t *uncompp)
338 {
339     dsl_deadlist_entry_t *dle;
340     dsl_deadlist_entry_t dle_tofind;
341     avl_index_t where;
342
343     if (dl->dl_oldfmt) {
344         VERIFY0(bpobj_space_range(&dl->dl_bpobj,
345             VERIFY3U(0, ==, bpobj_space_range(&dl->dl_bpobj,
346                 mintxg, maxtxg, usedp, compp, uncompp)));
347     }
348
349     *usedp = *compp = *uncompp = 0;
350
351     mutex_enter(&dl->dl_lock);
352     dsl_deadlist_load_tree(dl);
353     dle_tofind.dle_mintxg = mintxg;
354     dle = avl_find(&dl->dl_tree, &dle_tofind, &where);
355     /*
356      * If we don't find this mintxg, there shouldn't be anything
357      * after it either.
358      */
359     ASSERT(dle != NULL ||
360         avl_nearest(&dl->dl_tree, where, AVL_AFTER) == NULL);
361
362     for (; dle && dle->dle_mintxg < maxtxg;
363         dle = AVL_NEXT(&dl->dl_tree, dle)) {
364         uint64_t used, comp, uncomp;
365
366         VERIFY0(bpobj_space(&dle->dle_bpobj,
367             VERIFY3U(0, ==, bpobj_space(&dle->dle_bpobj,
368                 &used, &comp, &uncomp)));
369
370         *usedp += used;
371         *compp += comp;
372         *uncompp += uncomp;
373     }
374     mutex_exit(&dl->dl_lock);
375 }

```

```

376 static void
377 dsl_deadlist_insert_bpobj(dsl_deadlist_t *dl, uint64_t obj, uint64_t birth,
378     dmu_tx_t *tx)
379 {
380     dsl_deadlist_entry_t dle_tofind;
381     dsl_deadlist_entry_t *dle;
382     avl_index_t where;
383     uint64_t used, comp, uncomp;
384     bpobj_t bpo;
385
386     VERIFY0(bpobj_open(&bpo, dl->dl_os, obj));
387     VERIFY0(bpobj_space(&bpo, &used, &comp, &uncomp));
388     VERIFY3U(0, ==, bpobj_open(&bpo, dl->dl_os, obj));
389     VERIFY3U(0, ==, bpobj_space(&bpo, &used, &comp, &uncomp));
390     bpobj_close(&bpo);
391
392     dsl_deadlist_load_tree(dl);
393
394     dmu_buf_will_dirty(dl->dl_dbuf, tx);
395     mutex_enter(&dl->dl_lock);
396     dl->dl_phys->dl_used += used;
397     dl->dl_phys->dl_comp += comp;
398     dl->dl_phys->dl_uncomp += uncomp;
399     mutex_exit(&dl->dl_lock);
400
401     dle_tofind.dle_mintxg = birth;
402     dle = avl_find(&dl->dl_tree, &dle_tofind, &where);
403     if (dle == NULL)
404         dle = avl_nearest(&dl->dl_tree, where, AVL_BEFORE);
405     bpobj_enqueue_subobj(&dle->dle_bpobj, obj, tx);
406 }
407
408 unchanged_portion_omitted
409
410 /*
411  * Merge the deadlist pointed to by 'obj' into dl.  obj will be left as
412  * an empty deadlist.
413  */
414 void
415 dsl_deadlist_merge(dsl_deadlist_t *dl, uint64_t obj, dmu_tx_t *tx)
416 {
417     zap_cursor_t zc;
418     zap_attribute_t za;
419     dmu_buf_t *bonus;
420     dsl_deadlist_phys_t *dlp;
421     dmu_object_info_t doi;
422
423     VERIFY0(dmu_object_info(dl->dl_os, obj, &doi));
424     VERIFY3U(0, ==, dmu_object_info(dl->dl_os, obj, &doi));
425     if (doi.doi_type == DMU_OT_BPOBJ) {
426         bpobj_t bpo;
427         VERIFY0(bpobj_open(&bpo, dl->dl_os, obj));
428         VERIFY0(bpobj_iterate(&bpo,
429             VERIFY3U(0, ==, bpobj_open(&bpo, dl->dl_os, obj));
430             VERIFY3U(0, ==, bpobj_iterate(&bpo,
431                 dsl_deadlist_insert_cb, dl, tx)));
432         bpobj_close(&bpo);
433     }
434     return;
435 }
436
437 for (zap_cursor_init(&zc, dl->dl_os, obj);
438     zap_cursor_retrieve(&zc, &za) == 0;
439     zap_cursor_advance(&zc)) {
440     uint64_t mintxg = strtonum(za.za_name, NULL);
441     dsl_deadlist_insert_bpobj(dl, za.za_first_integer, mintxg, tx);
442     VERIFY0(zap_remove_int(dl->dl_os, obj, mintxg, tx));
443     VERIFY3U(0, ==, zap_remove_int(dl->dl_os, obj, mintxg, tx));

```

```

443     }
444     zap_cursor_fini(&ztc);

446     VERIFY0(dmu_bonus_hold(dl->dl_os, obj, FTAG, &bonus));
446     VERIFY3U(0, ==, dmu_bonus_hold(dl->dl_os, obj, FTAG, &bonus));
447     dlp = bonus->db_data;
448     dmu_buf_will_dirty(bonus, tx);
449     bzero(dlp, sizeof (*dlp));
450     dmu_buf_rele(bonus, FTAG);
451 }

453 /*
454  * Remove entries on dl that are >= mintxg, and put them on the bpobj.
455  */
456 void
457 dsl_deadlist_move_bpobj(dsl_deadlist_t *dl, bpobj_t *bpo, uint64_t mintxg,
458     dmu_tx_t *tx)
459 {
460     dsl_deadlist_entry_t dle_tofind;
461     dsl_deadlist_entry_t *dle;
462     avl_index_t where;

464     ASSERT(!dl->dl_oldfmt);
465     dmu_buf_will_dirty(dl->dl_dbuf, tx);
466     dsl_deadlist_load_tree(dl);

468     dle_tofind.dle_mintxg = mintxg;
469     dle = avl_find(&dl->dl_tree, &dle_tofind, &where);
470     if (dle == NULL)
471         dle = avl_nearest(&dl->dl_tree, where, AVL_AFTER);
472     while (dle) {
473         uint64_t used, comp, uncomp;
474         dsl_deadlist_entry_t *dle_next;

476         bpobj_enqueue_subobj(bpo, dle->dle_bpobj.bpo_object, tx);

478         VERIFY0(bpobj_space(&dle->dle_bpobj,
478             VERIFY3U(0, ==, bpobj_space(&dle->dle_bpobj,
479                 &used, &comp, &uncomp));
480         mutex_enter(&dl->dl_lock);
481         ASSERT3U(dl->dl_phys->dl_used, >=, used);
482         ASSERT3U(dl->dl_phys->dl_comp, >=, comp);
483         ASSERT3U(dl->dl_phys->dl_uncomp, >=, uncomp);
484         dl->dl_phys->dl_used -= used;
485         dl->dl_phys->dl_comp -= comp;
486         dl->dl_phys->dl_uncomp -= uncomp;
487         mutex_exit(&dl->dl_lock);

489         VERIFY0(zap_remove_int(dl->dl_os, dl->dl_object,
489             VERIFY3U(0, ==, zap_remove_int(dl->dl_os, dl->dl_object,
490                 dle->dle_mintxg, tx));

492         dle_next = AVL_NEXT(&dl->dl_tree, dle);
493         avl_remove(&dl->dl_tree, dle);
494         bpobj_close(&dle->dle_bpobj);
495         kmem_free(dle, sizeof (*dle));
496         dle = dle_next;
497     }
498 }

```

unchanged portion omitted

```

*****
36542 Thu Jul 26 14:19:15 2012
new/usr/src/uts/common/fs/zfs/dsl_dir.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

477 void
478 dsl_dir_destroy_sync(void *arg1, void *tag, dmu_tx_t *tx)
479 {
480     dsl_dir_t *dd = arg1;
481     objset_t *mos = dd->dd_pool->dp_meta_objset;
482     uint64_t obj;
483     dd_used_t t;

485     ASSERT(RW_WRITE_HELD(&dd->dd_pool->dp_config_rwlock));
486     ASSERT(dd->dd_phys->dd_head_dataset_obj == 0);

488     /*
489      * Remove our reservation. The impl() routine avoids setting the
490      * actual property, which would require the (already destroyed) ds.
491      */
492     dsl_dir_set_reservation_sync_impl(dd, 0, tx);

494     ASSERT0(dd->dd_phys->dd_used_bytes);
495     ASSERT0(dd->dd_phys->dd_reserved);
496     ASSERT3U(dd->dd_phys->dd_used_bytes, ==, 0);
497     ASSERT3U(dd->dd_phys->dd_reserved, ==, 0);
498     for (t = 0; t < DD_USED_NUM; t++)
499         ASSERT0(dd->dd_phys->dd_used_breakdown[t]);
500     ASSERT3U(dd->dd_phys->dd_used_breakdown[t], ==, 0);

501     VERIFY(0 == zap_destroy(mos, dd->dd_phys->dd_child_dir_zapobj, tx));
502     VERIFY(0 == zap_destroy(mos, dd->dd_phys->dd_props_zapobj, tx));
503     VERIFY(0 == dsl_deleg_destroy(mos, dd->dd_phys->dd_deleg_zapobj, tx));
504     VERIFY(0 == zap_remove(mos,
505         dd->dd_parent->dd_phys->dd_child_dir_zapobj, dd->dd_myname, tx));

505     obj = dd->dd_object;
506     dsl_dir_close(dd, tag);
507     VERIFY(0 == dmu_object_free(mos, obj, tx));
508 }
_____unchanged_portion_omitted_____

580 void
581 dsl_dir_sync(dsl_dir_t *dd, dmu_tx_t *tx)
582 {
583     ASSERT(dmu_tx_is_syncing(tx));

585     dmu_buf_will_dirty(dd->dd_dbuf, tx);

587     mutex_enter(&dd->dd_lock);
588     ASSERT0(dd->dd_temppreserved[tx->tx_txg&TXG_MASK]);
589     ASSERT3U(dd->dd_temppreserved[tx->tx_txg&TXG_MASK], ==, 0);
590     dprintf_dd(dd, "txg=%llu towrite=%lluK\n", tx->tx_txg,
591         dd->dd_space_towrite[tx->tx_txg&TXG_MASK] / 1024);
592     dd->dd_space_towrite[tx->tx_txg&TXG_MASK] = 0;
593     mutex_exit(&dd->dd_lock);

594     /* release the hold from dsl_dir_dirty */
595     dmu_buf_rele(dd->dd_dbuf, dd);
596 }
_____unchanged_portion_omitted_____

1294 static void
1295 dsl_dir_rename_sync(void *arg1, void *arg2, dmu_tx_t *tx)

```

```

1296 {
1297     dsl_dir_t *dd = arg1;
1298     struct renamearg *ra = arg2;
1299     dsl_pool_t *dp = dd->dd_pool;
1300     objset_t *mos = dp->dp_meta_objset;
1301     int err;
1302     char namebuf[MAXNAMELEN];

1304     ASSERT(dmu_buf_refcount(dd->dd_dbuf) <= 2);

1306     /* Log this before we change the name. */
1307     dsl_dir_name(ra->newparent, namebuf);
1308     spa_history_log_internal_dd(dd, "rename", tx,
1309         "-> %s/%s", namebuf, ra->mynewname);

1311     if (ra->newparent != dd->dd_parent) {
1312         dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD,
1313             -dd->dd_phys->dd_used_bytes,
1314             -dd->dd_phys->dd_compressed_bytes,
1315             -dd->dd_phys->dd_uncompressed_bytes, tx);
1316         dsl_dir_diduse_space(ra->newparent, DD_USED_CHILD,
1317             dd->dd_phys->dd_used_bytes,
1318             dd->dd_phys->dd_compressed_bytes,
1319             dd->dd_phys->dd_uncompressed_bytes, tx);

1321         if (dd->dd_phys->dd_reserved > dd->dd_phys->dd_used_bytes) {
1322             uint64_t unused_rsrv = dd->dd_phys->dd_reserved -
1323                 dd->dd_phys->dd_used_bytes;

1325             dsl_dir_diduse_space(dd->dd_parent, DD_USED_CHILD_RSRV,
1326                 -unused_rsrv, 0, 0, tx);
1327             dsl_dir_diduse_space(ra->newparent, DD_USED_CHILD_RSRV,
1328                 unused_rsrv, 0, 0, tx);
1329         }
1330     }

1332     dmu_buf_will_dirty(dd->dd_dbuf, tx);

1334     /* remove from old parent zapobj */
1335     err = zap_remove(mos, dd->dd_parent->dd_phys->dd_child_dir_zapobj,
1336         dd->dd_myname, tx);
1337     ASSERT0(err);
1338     ASSERT3U(err, ==, 0);

1339     (void) strcpy(dd->dd_myname, ra->mynewname);
1340     dsl_dir_close(dd->dd_parent, dd);
1341     dd->dd_phys->dd_parent_obj = ra->newparent->dd_object;
1342     VERIFY(0 == dsl_dir_open_obj(dd->dd_pool,
1343         ra->newparent->dd_object, NULL, dd, &dd->dd_parent));

1345     /* add to new parent zapobj */
1346     err = zap_add(mos, ra->newparent->dd_phys->dd_child_dir_zapobj,
1347         dd->dd_myname, 8, 1, &dd->dd_object, tx);
1348     ASSERT0(err);
1349     ASSERT3U(err, ==, 0);

1350 }
_____unchanged_portion_omitted_____

```

```

*****
23854 Thu Jul 26 14:19:16 2012
new/usr/src/uts/common/fs/zfs/dsl_pool.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

120 int
121 dsl_pool_open(dsl_pool_t *dp)
122 {
123     int err;
124     dsl_dir_t *dd;
125     dsl_dataset_t *ds;
126     uint64_t obj;

128     ASSERT(!dmu_objset_is_dirty_anywhere(dp->dp_meta_objset));

130     rw_enter(&dp->dp_config_rwlock, RW_WRITER);
131     err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
132                   DMU_POOL_ROOT_DATASET, sizeof (uint64_t), 1,
133                   &dp->dp_root_dir_obj);
134     if (err)
135         goto out;

137     err = dsl_dir_open_obj(dp, dp->dp_root_dir_obj,
138                          NULL, dp, &dp->dp_root_dir);
139     if (err)
140         goto out;

142     err = dsl_pool_open_special_dir(dp, MOS_DIR_NAME, &dp->dp_mos_dir);
143     if (err)
144         goto out;

146     if (spa_version(dp->dp_spa) >= SPA_VERSION_ORIGIN) {
147         err = dsl_pool_open_special_dir(dp, ORIGIN_DIR_NAME, &dd);
148         if (err)
149             goto out;
150         err = dsl_dataset_hold_obj(dp, dd->dd_phys->dd_head_dataset_obj,
151                                  FTAG, &ds);
152         if (err == 0) {
153             err = dsl_dataset_hold_obj(dp,
154                                       ds->ds_phys->ds_prev_snap_obj, dp,
155                                       &dp->dp_origin_snap);
156             dsl_dataset_rele(ds, FTAG);
157         }
158         dsl_dir_close(dd, dp);
159         if (err)
160             goto out;
161     }

163     if (spa_version(dp->dp_spa) >= SPA_VERSION_DEADLISTS) {
164         err = dsl_pool_open_special_dir(dp, FREE_DIR_NAME,
165                                       &dp->dp_free_dir);
166         if (err)
167             goto out;

169         err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
170                         DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj);
171         if (err)
172             goto out;
173         VERIFY0(bpobj_open(&dp->dp_free_bpobj,
174                          VERIFY3U(0, ==, bpobj_open(&dp->dp_free_bpobj,
175                                                       dp->dp_meta_objset, obj)));
175     }

177     if (spa_feature_is_active(dp->dp_spa,

```

```

178         &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
179             err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
180                             DMU_POOL_BPTREE_OBJ, sizeof (uint64_t), 1,
181                             &dp->dp_bptree_obj);
182             if (err != 0)
183                 goto out;
184         }

186     err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
187                   DMU_POOL_TMP_USERREFS, sizeof (uint64_t), 1,
188                   &dp->dp_tmp_userrefs_obj);
189     if (err == ENOENT)
190         err = 0;
191     if (err)
192         goto out;

194     err = dsl_scan_init(dp, dp->dp_tx.tx_open_txg);

196 out:
197     rw_exit(&dp->dp_config_rwlock);
198     return (err);
199 }
_____unchanged_portion_omitted_____

242 dsl_pool_t *
243 dsl_pool_create(spa_t *spa, nvlist_t *zplprops, uint64_t txg)
244 {
245     int err;
246     dsl_pool_t *dp = dsl_pool_open_impl(spa, txg);
247     dmu_tx_t *tx = dmu_tx_create_assigned(dp, txg);
248     objset_t *os;
249     dsl_dataset_t *ds;
250     uint64_t obj;

252     /* create and open the MOS (meta-objset) */
253     dp->dp_meta_objset = dmu_objset_create_impl(spa,
254                                               NULL, &dp->dp_meta_rootbp, DMU_OT_META, tx);

256     /* create the pool directory */
257     err = zap_create_claim(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
258                          DMU_OT_OBJECT_DIRECTORY, DMU_OT_NONE, 0, tx);
259     ASSERT0(err);
260     ASSERT3U(err, ==, 0);

261     /* Initialize scan structures */
262     VERIFY0(dsl_scan_init(dp, txg));
263     VERIFY3U(0, ==, dsl_scan_init(dp, txg));

264     /* create and open the root dir */
265     dp->dp_root_dir_obj = dsl_dir_create_sync(dp, NULL, NULL, tx);
266     VERIFY(0 == dsl_dir_open_obj(dp, dp->dp_root_dir_obj,
267                                 NULL, dp, &dp->dp_root_dir));

269     /* create and open the meta-objset dir */
270     (void) dsl_dir_create_sync(dp, dp->dp_root_dir, MOS_DIR_NAME, tx);
271     VERIFY(0 == dsl_pool_open_special_dir(dp,
272                                           MOS_DIR_NAME, &dp->dp_mos_dir));

274     if (spa_version(spa) >= SPA_VERSION_DEADLISTS) {
275         /* create and open the free dir */
276         (void) dsl_dir_create_sync(dp, dp->dp_root_dir,
277                                   FREE_DIR_NAME, tx);
278         VERIFY(0 == dsl_pool_open_special_dir(dp,
279                                               FREE_DIR_NAME, &dp->dp_free_dir));

281         /* create and open the free_bpobj */

```

```

282     obj = bpobj_alloc(dp->dp_meta_objset, SPA_MAXBLOCKSIZE, tx);
283     VERIFY(zap_add(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
284             DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj, tx) == 0);
285     VERIFY0(bpobj_open(&dp->dp_free_bpobj,
286             VERIFY3U(0, ==, bpobj_open(&dp->dp_free_bpobj,
287             dp->dp_meta_objset, obj)));
288 }
289
289 if (spa_version(spa) >= SPA_VERSION_DSL_SCRUB)
290     dsl_pool_create_origin(dp, tx);
291
292 /* create the root dataset */
293 obj = dsl_dataset_create_sync_dd(dp->dp_root_dir, NULL, 0, tx);
294
295 /* create the root objset */
296 VERIFY(0 == dsl_dataset_hold_obj(dp, obj, FTAG, &ds));
297 os = dmu_objset_create_impl(dp->dp_spa, ds,
298     dsl_dataset_get_blkptr(ds), DMU_OST_ZFS, tx);
299 #ifdef _KERNEL
300     zfs_create_fs(os, kcred, zplprops, tx);
301 #endif
302     dsl_dataset_rele(ds, FTAG);
303
304     dmu_tx_commit(tx);
305
306     return (dp);
307 }
308
309 unchanged_portion_omitted
310
311 void
312 dsl_pool_upgrade_clones(dsl_pool_t *dp, dmu_tx_t *tx)
313 {
314     ASSERT(dmu_tx_is_syncing(tx));
315     ASSERT(dp->dp_origin_snap != NULL);
316
317     VERIFY0(dmu_objset_find_spa(dp->dp_spa, NULL, upgrade_clones_cb,
318     VERIFY3U(0, ==, dmu_objset_find_spa(dp->dp_spa, NULL, upgrade_clones_cb,
319     tx, DS_FIND_CHILDREN));
320 }
321
322 /* ARGSUSED */
323 static int
324 upgrade_dir_clones_cb(spa_t *spa, uint64_t dsobj, const char *dsname, void *arg)
325 {
326     dmu_tx_t *tx = arg;
327     dsl_dataset_t *ds;
328     dsl_pool_t *dp = spa_get_dsl(spa);
329     objset_t *mos = dp->dp_meta_objset;
330
331     VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
332     VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
333
334     if (ds->ds_dir->dd_phys->dd_origin_obj) {
335         dsl_dataset_t *origin;
336
337         VERIFY0(dsl_dataset_hold_obj(dp,
338             VERIFY3U(0, ==, dsl_dataset_hold_obj(dp,
339             ds->ds_dir->dd_phys->dd_origin_obj, FTAG, &origin));
340
341         if (origin->ds_dir->dd_phys->dd_clones == 0) {
342             dmu_buf_will_dirty(origin->ds_dir->dd_dbuf, tx);
343             origin->ds_dir->dd_phys->dd_clones = zap_create(mos,
344                 DMU_OT_DSL_CLONES, DMU_OT_NONE, 0, tx);
345         }
346     }
347
348     VERIFY0(zap_add_int(dp->dp_meta_objset,

```

```

704     VERIFY3U(0, ==, zap_add_int(dp->dp_meta_objset,
705     origin->ds_dir->dd_phys->dd_clones, dsobj, tx));
706
707     dsl_dataset_rele(origin, FTAG);
708 }
709
710     dsl_dataset_rele(ds, FTAG);
711     return (0);
712 }
713
714 void
715 dsl_pool_upgrade_dir_clones(dsl_pool_t *dp, dmu_tx_t *tx)
716 {
717     ASSERT(dmu_tx_is_syncing(tx));
718     uint64_t obj;
719
720     (void) dsl_dir_create_sync(dp, dp->dp_root_dir, FREE_DIR_NAME, tx);
721     VERIFY(0 == dsl_pool_open_special_dir(dp,
722     FREE_DIR_NAME, &dp->dp_free_dir));
723
724     /*
725     * We can't use bpobj_alloc(), because spa_version() still
726     * returns the old version, and we need a new-version bpobj with
727     * subobj support. So call dmu_object_alloc() directly.
728     */
729     obj = dmu_object_alloc(dp->dp_meta_objset, DMU_OT_BPOBJ,
730     SPA_MAXBLOCKSIZE, DMU_OT_BPOBJ_HDR, sizeof (bpobj_phys_t), tx);
731     VERIFY0(zap_add(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
732     VERIFY3U(0, ==, zap_add(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
733     DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj, tx));
734     VERIFY0(bpobj_open(&dp->dp_free_bpobj,
735     VERIFY3U(0, ==, bpobj_open(&dp->dp_free_bpobj,
736     dp->dp_meta_objset, obj));
737
738     VERIFY0(dmu_objset_find_spa(dp->dp_spa, NULL,
739     VERIFY3U(0, ==, dmu_objset_find_spa(dp->dp_spa, NULL,
740     upgrade_dir_clones_cb, tx, DS_FIND_CHILDREN));
741 }
742
743 unchanged_portion_omitted

```

```

*****
50559 Thu Jul 26 14:19:16 2012
new/usr/src/uts/common/fs/zfs/dsl_scan.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

815 void
816 dsl_scan_ds_destroyed(dsl_dataset_t *ds, dmu_tx_t *tx)
817 {
818     dsl_pool_t *dp = ds->ds_dir->dd_pool;
819     dsl_scan_t *scn = dp->dp_scan;
820     uint64_t mintxg;

822     if (scn->scn_phys.scn_state != DSS_SCANNING)
823         return;

825     if (scn->scn_phys.scn_bookmark.zb_objset == ds->ds_object) {
826         if (dsl_dataset_is_snapshot(ds)) {
827             /* Note, scn_cur_{min,max}_txg stays the same. */
828             scn->scn_phys.scn_bookmark.zb_objset =
829                 ds->ds_phys->ds_next_snap_obj;
830             zfs_dbgmsg("destroying ds %llu; currently traversing; "
831                 "reset zb_objset to %llu",
832                 (u_longlong_t)ds->ds_object,
833                 (u_longlong_t)ds->ds_next_snap_obj);
834             scn->scn_phys.scn_flags |= DSF_VISIT_DS_AGAIN;
835         } else {
836             SET_BOOKMARK(&scn->scn_phys.scn_bookmark,
837                 ZB_DESTROYED_OBJSET, 0, 0, 0);
838             zfs_dbgmsg("destroying ds %llu; currently traversing; "
839                 "reset bookmark to -1,0,0,0",
840                 (u_longlong_t)ds->ds_object);
841         }
842     } else if (zap_lookup_int_key(dp->dp_meta_objset,
843         scn->scn_phys.scn_queue_obj, ds->ds_object, &mintxg) == 0) {
844         ASSERT3U(ds->ds_phys->ds_num_children, <=, 1);
845         VERIFY0(zap_remove_int(dp->dp_meta_objset,
846             VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
847                 scn->scn_phys.scn_queue_obj, ds->ds_object, tx));
848             if (dsl_dataset_is_snapshot(ds)) {
849                 /*
850                  * We keep the same mintxg; it could be >
851                  * ds_creation_txg if the previous snapshot was
852                  * deleted too.
853                  */
854                 VERIFY(zap_add_int_key(dp->dp_meta_objset,
855                     scn->scn_phys.scn_queue_obj,
856                     ds->ds_phys->ds_next_snap_obj, mintxg, tx) == 0);
857                 zfs_dbgmsg("destroying ds %llu; in queue; "
858                     "replacing with %llu",
859                     (u_longlong_t)ds->ds_object,
860                     (u_longlong_t)ds->ds_next_snap_obj);
861             } else {
862                 zfs_dbgmsg("destroying ds %llu; in queue; removing",
863                     (u_longlong_t)ds->ds_object);
864             }
865         } else {
866             zfs_dbgmsg("destroying ds %llu; ignoring",
867                 (u_longlong_t)ds->ds_object);
868         }
869     }
870     /*
871     * dsl_scan_sync() should be called after this, and should sync
872     * out our changed state, but just to be safe, do it here.
873     */

```

```

873     dsl_scan_sync_state(scn, tx);
874 }

876 void
877 dsl_scan_ds_snapshotted(dsl_dataset_t *ds, dmu_tx_t *tx)
878 {
879     dsl_pool_t *dp = ds->ds_dir->dd_pool;
880     dsl_scan_t *scn = dp->dp_scan;
881     uint64_t mintxg;

883     if (scn->scn_phys.scn_state != DSS_SCANNING)
884         return;

886     ASSERT(ds->ds_phys->ds_prev_snap_obj != 0);

888     if (scn->scn_phys.scn_bookmark.zb_objset == ds->ds_object) {
889         scn->scn_phys.scn_bookmark.zb_objset =
890             ds->ds_phys->ds_prev_snap_obj;
891         zfs_dbgmsg("snapshotting ds %llu; currently traversing; "
892             "reset zb_objset to %llu",
893             (u_longlong_t)ds->ds_object,
894             (u_longlong_t)ds->ds_phys->ds_prev_snap_obj);
895     } else if (zap_lookup_int_key(dp->dp_meta_objset,
896         scn->scn_phys.scn_queue_obj, ds->ds_object, &mintxg) == 0) {
897         VERIFY0(zap_remove_int(dp->dp_meta_objset,
898             VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
899                 scn->scn_phys.scn_queue_obj, ds->ds_object, tx));
900             VERIFY(zap_add_int_key(dp->dp_meta_objset,
901                 scn->scn_phys.scn_queue_obj,
902                 ds->ds_phys->ds_prev_snap_obj, mintxg, tx) == 0);
903             zfs_dbgmsg("snapshotting ds %llu; in queue; "
904                 "replacing with %llu",
905                 (u_longlong_t)ds->ds_object,
906                 (u_longlong_t)ds->ds_phys->ds_prev_snap_obj);
907         }
908     }
909     dsl_scan_sync_state(scn, tx);
910 }

911 void
912 dsl_scan_ds_clone_swapped(dsl_dataset_t *ds1, dsl_dataset_t *ds2, dmu_tx_t *tx)
913 {
914     dsl_pool_t *dp = ds1->ds_dir->dd_pool;
915     dsl_scan_t *scn = dp->dp_scan;
916     uint64_t mintxg;

917     if (scn->scn_phys.scn_state != DSS_SCANNING)
918         return;

920     if (scn->scn_phys.scn_bookmark.zb_objset == ds1->ds_object) {
921         scn->scn_phys.scn_bookmark.zb_objset = ds2->ds_object;
922         zfs_dbgmsg("clone_swap ds %llu; currently traversing; "
923             "reset zb_objset to %llu",
924             (u_longlong_t)ds1->ds_object,
925             (u_longlong_t)ds2->ds_object);
926     } else if (scn->scn_phys.scn_bookmark.zb_objset == ds2->ds_object) {
927         scn->scn_phys.scn_bookmark.zb_objset = ds1->ds_object;
928         zfs_dbgmsg("clone_swap ds %llu; currently traversing; "
929             "reset zb_objset to %llu",
930             (u_longlong_t)ds2->ds_object,
931             (u_longlong_t)ds1->ds_object);
932     }

934     if (zap_lookup_int_key(dp->dp_meta_objset, scn->scn_phys.scn_queue_obj,
935         ds1->ds_object, &mintxg) == 0) {
936         int err;

```

```

938     ASSERT3U(mintxg, ==, dsl->ds_phys->ds_prev_snap_txg);
939     ASSERT3U(mintxg, ==, ds2->ds_phys->ds_prev_snap_txg);
940     VERIFY0(zap_remove_int(dp->dp_meta_objset,
940     VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
941     scn->scn_phys.scn_queue_obj, dsl->ds_object, tx));
942     err = zap_add_int_key(dp->dp_meta_objset,
943     scn->scn_phys.scn_queue_obj, ds2->ds_object, mintxg, tx);
944     VERIFY(err == 0 || err == EEXIST);
945     if (err == EEXIST) {
946         /* Both were there to begin with */
947         VERIFY(0 == zap_add_int_key(dp->dp_meta_objset,
948     scn->scn_phys.scn_queue_obj,
949     dsl->ds_object, mintxg, tx));
950     }
951     zfs_dbgmsg("clone_swap ds %llu; in queue; "
952     "replacing with %llu",
953     (u_longlong_t)dsl->ds_object,
954     (u_longlong_t)ds2->ds_object);
955     } else if (zap_lookup_int_key(dp->dp_meta_objset,
956     scn->scn_phys.scn_queue_obj, ds2->ds_object, &mintxg) == 0) {
957     ASSERT3U(mintxg, ==, dsl->ds_phys->ds_prev_snap_txg);
958     ASSERT3U(mintxg, ==, ds2->ds_phys->ds_prev_snap_txg);
959     VERIFY0(zap_remove_int(dp->dp_meta_objset,
959     VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
960     scn->scn_phys.scn_queue_obj, ds2->ds_object, tx));
961     VERIFY(0 == zap_add_int_key(dp->dp_meta_objset,
962     scn->scn_phys.scn_queue_obj, dsl->ds_object, mintxg, tx));
963     zfs_dbgmsg("clone_swap ds %llu; in queue; "
964     "replacing with %llu",
965     (u_longlong_t)ds2->ds_object,
966     (u_longlong_t)dsl->ds_object);
967     }
968     dsl_scan_sync_state(scn, tx);
969 }
970 }
unchanged portion omitted
1010 static void
1011 dsl_scan_visitds(dsl_scan_t *scn, uint64_t dsobj, dmu_tx_t *tx)
1012 {
1013     dsl_pool_t *dp = scn->scn_dp;
1014     dsl_dataset_t *ds;
1015     objset_t *os;
1016
1017     VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
1018     VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
1019
1020     if (dmu_objset_from_ds(ds, &os))
1021         goto out;
1022
1023     /*
1024     * Only the ZIL in the head (non-snapshot) is valid. Even though
1025     * snapshots can have ZIL block pointers (which may be the same
1026     * BP as in the head), they must be ignored. So we traverse the
1027     * ZIL here, rather than in scan_recurse(), because the regular
1028     * snapshot block-sharing rules don't apply to it.
1029     */
1030     if (DSL_SCAN_IS_SCRUB_RESILVER(scn) && !dsl_dataset_is_snapshot(ds))
1031         dsl_scan_zil(dp, &os->os_zil_header);
1032
1033     /*
1034     * Iterate over the bps in this ds.
1035     */
1036     dmu_buf_will_dirty(ds->ds_dbuf, tx);
1037     dsl_scan_visit_rootbp(scn, ds, &ds->ds_phys->ds_bp, tx);

```

```

1038     char *dsname = kmem_alloc(ZFS_MAXNAMELEN, KM_SLEEP);
1039     dsl_dataset_name(ds, dsname);
1040     zfs_dbgmsg("scanned dataset %llu (%s) with min=%llu max=%llu; "
1041     "pausing=%u",
1042     (longlong_t)dsobj, dsname,
1043     (longlong_t)scn->scn_phys.scn_cur_min_txg,
1044     (longlong_t)scn->scn_phys.scn_cur_max_txg,
1045     (int)scn->scn_pausing);
1046     kmem_free(dsname, ZFS_MAXNAMELEN);
1047
1048     if (scn->scn_pausing)
1049         goto out;
1050
1051     /*
1052     * We've finished this pass over this dataset.
1053     */
1054
1055     /*
1056     * If we did not completely visit this dataset, do another pass.
1057     */
1058     if (scn->scn_phys.scn_flags & DSF_VISIT_DS_AGAIN) {
1059         zfs_dbgmsg("incomplete pass; visiting again");
1060         scn->scn_phys.scn_flags &= ~DSF_VISIT_DS_AGAIN;
1061         VERIFY(zap_add_int_key(dp->dp_meta_objset,
1062     scn->scn_phys.scn_queue_obj, ds->ds_object,
1063     scn->scn_phys.scn_cur_max_txg, tx) == 0);
1064         goto out;
1065     }
1066
1067     /*
1068     * Add descendent datasets to work queue.
1069     */
1070     if (ds->ds_phys->ds_next_snap_obj != 0) {
1071         VERIFY(zap_add_int_key(dp->dp_meta_objset,
1072     scn->scn_phys.scn_queue_obj, ds->ds_phys->ds_next_snap_obj,
1073     ds->ds_phys->ds_creation_txg, tx) == 0);
1074     }
1075     if (ds->ds_phys->ds_num_children > 1) {
1076         boolean_t usenext = B_FALSE;
1077         if (ds->ds_phys->ds_next_clones_obj != 0) {
1078             uint64_t count;
1079             /*
1080             * A bug in a previous version of the code could
1081             * cause upgrade_clones_cb() to not set
1082             * ds_next_snap_obj when it should, leading to a
1083             * missing entry. Therefore we can only use the
1084             * next_clones_obj when its count is correct.
1085             */
1086             int err = zap_count(dp->dp_meta_objset,
1087     ds->ds_phys->ds_next_clones_obj, &count);
1088             if (err == 0 &&
1089     count == ds->ds_phys->ds_num_children - 1)
1090                 usenext = B_TRUE;
1091         }
1092     }
1093     if (usenext) {
1094         VERIFY(zap_join_key(dp->dp_meta_objset,
1095     ds->ds_phys->ds_next_clones_obj,
1096     scn->scn_phys.scn_queue_obj,
1097     ds->ds_phys->ds_creation_txg, tx) == 0);
1098     } else {
1099         struct enqueue_clones_arg eca;
1100         eca.tx = tx;
1101         eca.originobj = ds->ds_object;
1102     }
1103     (void) dmu_objset_find_spa(ds->ds_dir->dd_pool->dp_spa,

```

```

1104             NULL, enqueue_clones_cb, &eca, DS_FIND_CHILDREN);
1105         }
1106     }

1108 out:
1109     dsl_dataset_rele(ds, FTAG);
1110 }
unchanged portion omitted

1249 static void
1250 dsl_scan_visit(dsl_scan_t *scn, dmu_tx_t *tx)
1251 {
1252     dsl_pool_t *dp = scn->scn_dp;
1253     zap_cursor_t zc;
1254     zap_attribute_t za;

1256     if (scn->scn_phys.scn_ddt_bookmark.ddb_class <=
1257         scn->scn_phys.scn_ddt_class_max) {
1258         scn->scn_phys.scn_cur_min_txg = scn->scn_phys.scn_min_txg;
1259         scn->scn_phys.scn_cur_max_txg = scn->scn_phys.scn_max_txg;
1260         dsl_scan_ddt(scn, tx);
1261         if (scn->scn_pausing)
1262             return;
1263     }

1265     if (scn->scn_phys.scn_bookmark.zb_objset == DMU_META_OBJSET) {
1266         /* First do the MOS & ORIGIN */

1268         scn->scn_phys.scn_cur_min_txg = scn->scn_phys.scn_min_txg;
1269         scn->scn_phys.scn_cur_max_txg = scn->scn_phys.scn_max_txg;
1270         dsl_scan_visit_rootbp(scn, NULL,
1271             &dp->dp_meta_rootbp, tx);
1272         spa_set_rootblkptr(dp->dp_spa, &dp->dp_meta_rootbp);
1273         if (scn->scn_pausing)
1274             return;

1276         if (spa_version(dp->dp_spa) < SPA_VERSION_DSL_SCRUB) {
1277             VERIFY0 == dmu_objset_find_spa(dp->dp_spa,
1278                 NULL, enqueue_cb, tx, DS_FIND_CHILDREN));
1279         } else {
1280             dsl_scan_visitds(scn,
1281                 dp->dp_origin_snap->ds_object, tx);
1282         }
1283         ASSERT(!scn->scn_pausing);
1284     } else if (scn->scn_phys.scn_bookmark.zb_objset !=
1285         ZB_DESTROYED_OBJSET) {
1286         /*
1287          * If we were paused, continue from here. Note if the
1288          * ds were paused on was deleted, the zb_objset may
1289          * be -1, so we will skip this and find a new objset
1290          * below.
1291          */
1292         dsl_scan_visitds(scn, scn->scn_phys.scn_bookmark.zb_objset, tx);
1293         if (scn->scn_pausing)
1294             return;
1295     }

1297     /*
1298      * In case we were paused right at the end of the ds, zero the
1299      * bookmark so we don't think that we're still trying to resume.
1300      */
1301     bzero(&scn->scn_phys.scn_bookmark, sizeof (zbookmark_t));

1303     /* keep pulling things out of the zap-object-as-queue */
1304     while (zap_cursor_init(&zc, dp->dp_meta_objset,
1305         scn->scn_phys.scn_queue_obj),

```

```

1306         zap_cursor_retrieve(&zc, &za) == 0) {
1307         dsl_dataset_t *ds;
1308         uint64_t dsobj;

1310         dsobj = strtonum(za.za_name, NULL);
1311         VERIFY0(zap_remove_int(dp->dp_meta_objset,
1312             VERIFY3U(0, ==, zap_remove_int(dp->dp_meta_objset,
1313                 scn->scn_phys.scn_queue_obj, dsobj, tx)));

1314         /* Set up min/max txg */
1315         VERIFY0(dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
1316         VERIFY3U(0, ==, dsl_dataset_hold_obj(dp, dsobj, FTAG, &ds));
1317         if (za.za_first_integer != 0) {
1318             scn->scn_phys.scn_cur_min_txg =
1319                 MAX(scn->scn_phys.scn_min_txg,
1320                     za.za_first_integer);
1321         } else {
1322             scn->scn_phys.scn_cur_min_txg =
1323                 MAX(scn->scn_phys.scn_min_txg,
1324                     ds->ds_phys->ds_prev_snap_txg);
1325         }
1326         scn->scn_phys.scn_cur_max_txg = dsl_scan_ds_maxtxg(ds);
1327         dsl_dataset_rele(ds, FTAG);

1328         dsl_scan_visitds(scn, dsobj, tx);
1329         zap_cursor_fini(&zc);
1330         if (scn->scn_pausing)
1331             return;
1332     }
1333     zap_cursor_fini(&zc);
1334 }
unchanged portion omitted

1393 void
1394 dsl_scan_sync(dsl_pool_t *dp, dmu_tx_t *tx)
1395 {
1396     dsl_scan_t *scn = dp->dp_scan;
1397     spa_t *spa = dp->dp_spa;
1398     int err;

1400     /*
1401      * Check for scn_restart_txg before checking spa_load_state, so
1402      * that we can restart an old-style scan while the pool is being
1403      * imported (see dsl_scan_init).
1404      */
1405     if (scn->scn_restart_txg != 0 &&
1406         scn->scn_restart_txg <= tx->tx_txg) {
1407         pool_scan_func_t func = POOL_SCAN_SCRUB;
1408         dsl_scan_done(scn, B_FALSE, tx);
1409         if (vdev_resilver_needed(spa->spa_root_vdev, NULL, NULL))
1410             func = POOL_SCAN_RESILVER;
1411         zfs_dbgmsg("restarting scan func=%u txg=%llu",
1412             func, tx->tx_txg);
1413         dsl_scan_setup_sync(scn, &func, tx);
1414     }

1416     if (!dsl_scan_active(scn) ||
1417         spa_sync_pass(dp->dp_spa) > 1)
1418         return;

1420     scn->scn_visited_this_txg = 0;
1421     scn->scn_pausing = B_FALSE;
1422     scn->scn_sync_start_time = gethrtime();
1423     spa->spa_scrub_active = B_TRUE;

1425     /*

```



```

1426     * First process the free list.  If we pause the free, don't do
1427     * any scanning.  This ensures that there is no free list when
1428     * we are scanning, so the scan code doesn't have to worry about
1429     * traversing it.
1430     */
1431     if (spa_version(dp->dp_spa) >= SPA_VERSION_DEADLISTS) {
1432         scn->scn_is_bptree = B_FALSE;
1433         scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1434             NULL, ZIO_FLAG_MUSTSUCCEED);
1435         err = bpobj_iterate(&dp->dp_free_bpobj,
1436             dsl_scan_free_block_cb, scn, tx);
1437         VERIFY0(zio_wait(scn->scn_zio_root));
1438         VERIFY3U(0, ==, zio_wait(scn->scn_zio_root));
1439
1440         if (err == 0 && spa_feature_is_active(spa,
1441             &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
1442             scn->scn_is_bptree = B_TRUE;
1443             scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1444                 NULL, ZIO_FLAG_MUSTSUCCEED);
1445             err = bptree_iterate(dp->dp_meta_objset,
1446                 dp->dp_bptree_obj, B_TRUE, dsl_scan_free_block_cb,
1447                 scn, tx);
1448             VERIFY0(zio_wait(scn->scn_zio_root));
1449             VERIFY3U(0, ==, zio_wait(scn->scn_zio_root));
1450             if (err != 0)
1451                 return;
1452
1453             /* disable async destroy feature */
1454             spa_feature_decr(spa,
1455                 &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY], tx);
1456             ASSERT(!spa_feature_is_active(spa,
1457                 &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY]));
1458             VERIFY0(zap_remove(dp->dp_meta_objset,
1459                 DMU_POOL_DIRECTORY_OBJECT,
1460                 DMU_POOL_BPTREE_OBJ, tx));
1461             VERIFY0(bptree_free(dp->dp_meta_objset,
1462                 VERIFY3U(0, ==, bptree_free(dp->dp_meta_objset,
1463                     dp->dp_bptree_obj, tx));
1464                 dp->dp_bptree_obj = 0;
1465             }
1466             if (scn->scn_visited_this_txg) {
1467                 zfs_dbgmsg("freed %llu blocks in %llums from "
1468                     "free_bpobj/bptree txg %llu",
1469                     (longlong_t)scn->scn_visited_this_txg,
1470                     (longlong_t)
1471                     (gethrtime() - scn->scn_sync_start_time) / MICROSEC,
1472                     (longlong_t)tx->tx_txg);
1473                 scn->scn_visited_this_txg = 0;
1474                 /*
1475                  * Re-sync the ddt so that we can further modify
1476                  * it when doing bprewrite.
1477                  */
1478                 ddt_sync(spa, tx->tx_txg);
1479             }
1480             if (err == ERESTART)
1481                 return;
1482         }
1483
1484         if (scn->scn_phys.scn_state != DSS_SCANNING)
1485             return;
1486
1487         if (scn->scn_phys.scn_ddt_bookmark.ddb_class <=
1488             scn->scn_phys.scn_ddt_class_max) {
1489             zfs_dbgmsg("doing scan sync txg %llu; "
1490                 "ddt bm=%llu/%llu/%llu/%llx",

```

```

1488             (longlong_t)tx->tx_txg,
1489             (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_class,
1490             (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_type,
1491             (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_checksum,
1492             (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_cursor);
1493             ASSERT(scn->scn_phys.scn_bookmark.zb_objset == 0);
1494             ASSERT(scn->scn_phys.scn_bookmark.zb_object == 0);
1495             ASSERT(scn->scn_phys.scn_bookmark.zb_level == 0);
1496             ASSERT(scn->scn_phys.scn_bookmark.zb_blkid == 0);
1497         } else {
1498             zfs_dbgmsg("doing scan sync txg %llu; bm=%llu/%llu/%llu/%llu",
1499                 (longlong_t)tx->tx_txg,
1500                 (longlong_t)scn->scn_phys.scn_bookmark.zb_objset,
1501                 (longlong_t)scn->scn_phys.scn_bookmark.zb_object,
1502                 (longlong_t)scn->scn_phys.scn_bookmark.zb_level,
1503                 (longlong_t)scn->scn_phys.scn_bookmark.zb_blkid);
1504         }
1505
1506         scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1507             NULL, ZIO_FLAG_CANFAIL);
1508         dsl_scan_visit(scn, tx);
1509         (void) zio_wait(scn->scn_zio_root);
1510         scn->scn_zio_root = NULL;
1511
1512         zfs_dbgmsg("visited %llu blocks in %llums",
1513             (longlong_t)scn->scn_visited_this_txg,
1514             (longlong_t)(gethrtime() - scn->scn_sync_start_time) / MICROSEC);
1515
1516         if (!scn->scn_pausing) {
1517             /* finished with scan. */
1518             zfs_dbgmsg("finished scan txg %llu", (longlong_t)tx->tx_txg);
1519             dsl_scan_done(scn, B_TRUE, tx);
1520         }
1521
1522         if (DSL_SCAN_IS_SCRUB_RESILVER(scn)) {
1523             mutex_enter(&spa->spa_scrub_lock);
1524             while (spa->spa_scrub_inflight > 0) {
1525                 cv_wait(&spa->spa_scrub_io_cv,
1526                     &spa->spa_scrub_lock);
1527             }
1528             mutex_exit(&spa->spa_scrub_lock);
1529         }
1530
1531         dsl_scan_sync_state(scn, tx);
1532     }

```

\_\_\_\_\_unchanged portion omitted\_\_\_\_\_

```
*****
6280 Thu Jul 26 14:19:17 2012
new/usr/src/uts/common/fs/zfs/dsl_synctask.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

159 void
160 dsl_sync_task_group_sync(dsl_sync_task_group_t *dstg, dmu_tx_t *tx)
161 {
162     dsl_sync_task_t *dst;
163     dsl_pool_t *dp = dstg->dstg_pool;
164     uint64_t quota, used;

166     ASSERT0(dstg->dstg_err);
166     ASSERT3U(dstg->dstg_err, ==, 0);

168     /*
169     * Check for sufficient space. We just check against what's
170     * on-disk; we don't want any in-flight accounting to get in our
171     * way, because open context may have already used up various
172     * in-core limits (arc_temppreserve, dsl_pool_temppreserve).
173     */
174     quota = dsl_pool_adjustedsize(dp, B_FALSE) -
175         metaslab_class_get_deferred(spa_normal_class(dp->dp_spa));
176     used = dp->dp_root_dir->dd_phys->dd_used_bytes;
177     /* MOS space is triple-dittoed, so we multiply by 3. */
178     if (dstg->dstg_space > 0 && used + dstg->dstg_space * 3 > quota) {
179         dstg->dstg_err = ENOSPC;
180         return;
181     }

183     /*
184     * Check for errors by calling checkfuncs.
185     */
186     rw_enter(&dp->dp_config_rwlock, RW_WRITER);
187     for (dst = list_head(&dstg->dstg_tasks); dst;
188         dst = list_next(&dstg->dstg_tasks, dst)) {
189         dst->dst_err =
190             dst->dst_checkfunc(dst->dst_arg1, dst->dst_arg2, tx);
191         if (dst->dst_err)
192             dstg->dstg_err = dst->dst_err;
193     }

195     if (dstg->dstg_err == 0) {
196         /*
197         * Execute sync tasks.
198         */
199         for (dst = list_head(&dstg->dstg_tasks); dst;
200             dst = list_next(&dstg->dstg_tasks, dst)) {
201             dst->dst_syncfunc(dst->dst_arg1, dst->dst_arg2, tx);
202         }
203     }
204     rw_exit(&dp->dp_config_rwlock);

206     if (dstg->dstg_nowaiter)
207         dsl_sync_task_group_destroy(dstg);
208 }
_____unchanged_portion_omitted_____
```

```
*****
43720 Thu Jul 26 14:19:18 2012
new/usr/src/uts/common/fs/zfs/metaslub.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_

749 void
750 metaslab_fini(metaslab_t *msp)
751 {
752     metaslab_group_t *mg = msp->ms_group;

754     vdev_space_update(mg->mg_vd,
755                     -msp->ms_smo.smo_alloc, 0, -msp->ms_map.sm_size);

757     metaslab_group_remove(mg, msp);

759     mutex_enter(&msp->ms_lock);

761     space_map_unload(&msp->ms_map);
762     space_map_destroy(&msp->ms_map);

764     for (int t = 0; t < TXG_SIZE; t++) {
765         space_map_destroy(&msp->ms_allocmap[t]);
766         space_map_destroy(&msp->ms_freemap[t]);
767     }

769     for (int t = 0; t < TXG_DEFER_SIZE; t++)
770         space_map_destroy(&msp->ms_defermap[t]);

772     ASSERT0(msp->ms_deferspace);
772     ASSERT3S(msp->ms_deferspace, ==, 0);

774     mutex_exit(&msp->ms_lock);
775     mutex_destroy(&msp->ms_lock);

777     kmem_free(msp, sizeof (metaslub_t));
778 }
_____unchanged_portion_omitted_
```

```

*****
169821 Thu Jul 26 14:19:19 2012
new/usr/src/uts/common/fs/zfs/spa.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

2942 static void
2943 spa_add_feature_stats(spa_t *spa, nvlist_t *config)
2944 {
2945     nvlist_t *features;
2946     zap_cursor_t zc;
2947     zap_attribute_t za;

2949     ASSERT(spa_config_held(spa, SCL_CONFIG, RW_READER));
2950     VERIFY(nvlist_alloc(&features, NV_UNIQUE_NAME, KM_SLEEP) == 0);

2952     if (spa->spa_feat_for_read_obj != 0) {
2953         for (zap_cursor_init(&zc, spa->spa_meta_objset,
2954             spa->spa_feat_for_read_obj);
2955             zap_cursor_retrieve(&zc, &za) == 0;
2956             zap_cursor_advance(&zc)) {
2957             ASSERT(za.za_integer_length == sizeof (uint64_t) &&
2958                 za.za_num_integers == 1);
2959             VERIFY0(nvlist_add_uint64(features, za.za_name,
2960                 VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
2961                     za.za_first_integer)));
2962             zap_cursor_fini(&zc);
2963         }

2965     if (spa->spa_feat_for_write_obj != 0) {
2966         for (zap_cursor_init(&zc, spa->spa_meta_objset,
2967             spa->spa_feat_for_write_obj);
2968             zap_cursor_retrieve(&zc, &za) == 0;
2969             zap_cursor_advance(&zc)) {
2970             ASSERT(za.za_integer_length == sizeof (uint64_t) &&
2971                 za.za_num_integers == 1);
2972             VERIFY0(nvlist_add_uint64(features, za.za_name,
2973                 VERIFY3U(0, ==, nvlist_add_uint64(features, za.za_name,
2974                     za.za_first_integer)));
2975             zap_cursor_fini(&zc);
2976         }

2978     VERIFY(nvlist_add_nvlist(config, ZPOOL_CONFIG_FEATURE_STATS,
2979         features) == 0);
2980     nvlist_free(features);
2981 }
_____unchanged_portion_omitted_____

3224 /*
3225  * Pool Creation
3226  */
3227 int
3228 spa_create(const char *pool, nvlist_t *nvroot, nvlist_t *props,
3229     nvlist_t *zplprops)
3230 {
3231     spa_t *spa;
3232     char *altroot = NULL;
3233     vdev_t *rvd;
3234     dsl_pool_t *dp;
3235     dmu_tx_t *tx;
3236     int error = 0;
3237     uint64_t txg = TXG_INITIAL;
3238     nvlist_t **spares, **l2cache;

```

```

3239     uint_t nspares, nl2cache;
3240     uint64_t version, obj;
3241     boolean_t has_features;

3243     /*
3244      * If this pool already exists, return failure.
3245      */
3246     mutex_enter(&spa_namespace_lock);
3247     if (spa_lookup(pool) != NULL) {
3248         mutex_exit(&spa_namespace_lock);
3249         return (EEXIST);
3250     }

3252     /*
3253      * Allocate a new spa_t structure.
3254      */
3255     (void) nvlist_lookup_string(props,
3256         zpool_prop_to_name(ZPOOL_PROP_ALTROOT), &altroot);
3257     spa = spa_add(pool, NULL, altroot);
3258     spa_activate(spa, spa_mode_global);

3260     if (props && (error = spa_prop_validate(spa, props))) {
3261         spa_deactivate(spa);
3262         spa_remove(spa);
3263         mutex_exit(&spa_namespace_lock);
3264         return (error);
3265     }

3267     has_features = B_FALSE;
3268     for (nvpair_t *elem = nvlist_next_nvpair(props, NULL);
3269         elem != NULL; elem = nvlist_next_nvpair(props, elem)) {
3270         if (zpool_prop_feature(nvpair_name(elem)))
3271             has_features = B_TRUE;
3272     }

3274     if (has_features || nvlist_lookup_uint64(props,
3275         zpool_prop_to_name(ZPOOL_PROP_VERSION), &version) != 0) {
3276         version = SPA_VERSION;
3277     }
3278     ASSERT(SPA_VERSION_IS_SUPPORTED(version));

3280     spa->spa_first_txg = txg;
3281     spa->spa_uberblock.ub_txg = txg - 1;
3282     spa->spa_uberblock.ub_version = version;
3283     spa->spa_ubsync = spa->spa_uberblock;

3285     /*
3286      * Create "The Godfather" zio to hold all async IOs
3287      */
3288     spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
3289         ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);

3291     /*
3292      * Create the root vdev.
3293      */
3294     spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);

3296     error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, VDEV_ALLOC_ADD);

3298     ASSERT(error != 0 || rvd != NULL);
3299     ASSERT(error != 0 || spa->spa_root_vdev == rvd);

3301     if (error == 0 && !zfs_allocatable_devs(nvroot))
3302         error = EINVAL;

3304     if (error == 0 &&

```

```

3305     (error = vdev_create(rvd, txg, B_FALSE)) == 0 &&
3306     (error = spa_validate_aux(spa, nvroot, txg,
3307     VDEV_ALLOC_ADD) == 0) {
3308         for (int c = 0; c < rvd->vdev_children; c++) {
3309             vdev metaslab_set_size(rvd->vdev_child[c]);
3310             vdev_expand(rvd->vdev_child[c], txg);
3311         }
3312     }
3314     spa_config_exit(spa, SCL_ALL, FTAG);
3316     if (error != 0) {
3317         spa_unload(spa);
3318         spa_deactivate(spa);
3319         spa_remove(spa);
3320         mutex_exit(&spa_namespace_lock);
3321         return (error);
3322     }
3324     /*
3325     * Get the list of spares, if specified.
3326     */
3327     if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_SPARES,
3328     &spares, &nspares) == 0) {
3329         VERIFY(nvlist_alloc(&spa->spa_spares.sav_config, NV_UNIQUE_NAME,
3330         KM_SLEEP) == 0);
3331         VERIFY(nvlist_add_nvlist_array(spa->spa_spares.sav_config,
3332         ZPOOL_CONFIG_SPARES, spares, nspares) == 0);
3333         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3334         spa_load_spares(spa);
3335         spa_config_exit(spa, SCL_ALL, FTAG);
3336         spa->spa_spares.sav_sync = B_TRUE;
3337     }
3339     /*
3340     * Get the list of level 2 cache devices, if specified.
3341     */
3342     if (nvlist_lookup_nvlist_array(nvroot, ZPOOL_CONFIG_L2CACHE,
3343     &l2cache, &nl2cache) == 0) {
3344         VERIFY(nvlist_alloc(&spa->spa_l2cache.sav_config,
3345         NV_UNIQUE_NAME, KM_SLEEP) == 0);
3346         VERIFY(nvlist_add_nvlist_array(spa->spa_l2cache.sav_config,
3347         ZPOOL_CONFIG_L2CACHE, l2cache, nl2cache) == 0);
3348         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
3349         spa_load_l2cache(spa);
3350         spa_config_exit(spa, SCL_ALL, FTAG);
3351         spa->spa_l2cache.sav_sync = B_TRUE;
3352     }
3354     spa->spa_is_initializing = B_TRUE;
3355     spa->spa_dsl_pool = dp = dsl_pool_create(spa, zplprops, txg);
3356     spa->spa_meta_objset = dp->dp_meta_objset;
3357     spa->spa_is_initializing = B_FALSE;
3359     /*
3360     * Create DDTs (dedup tables).
3361     */
3362     ddt_create(spa);
3364     spa_update_dspace(spa);
3366     tx = dmu_tx_create_assigned(dp, txg);
3368     /*
3369     * Create the pool config object.
3370     */

```

```

3371     spa->spa_config_object = dmu_object_alloc(spa->spa_meta_objset,
3372     DMU_OT_PACKED_NVLIST, SPA_CONFIG_BLOCKSIZE,
3373     DMU_OT_PACKED_NVLIST_SIZE, sizeof (uint64_t), tx);
3375     if (zap_add(spa->spa_meta_objset,
3376     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CONFIG,
3377     sizeof (uint64_t), 1, &spa->spa_config_object, tx) != 0) {
3378         cmn_err(CE_PANIC, "failed to add pool config");
3379     }
3381     if (spa_version(spa) >= SPA_VERSION_FEATURES)
3382         spa_feature_create_zap_objects(spa, tx);
3384     if (zap_add(spa->spa_meta_objset,
3385     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_CREATION_VERSION,
3386     sizeof (uint64_t), 1, &version, tx) != 0) {
3387         cmn_err(CE_PANIC, "failed to add pool version");
3388     }
3390     /* Newly created pools with the right version are always deflated. */
3391     if (version >= SPA_VERSION_RAIDZ_DEFLATE) {
3392         spa->spa_deflate = TRUE;
3393         if (zap_add(spa->spa_meta_objset,
3394         DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
3395         sizeof (uint64_t), 1, &spa->spa_deflate, tx) != 0) {
3396             cmn_err(CE_PANIC, "failed to add deflate");
3397         }
3398     }
3400     /*
3401     * Create the deferred-free bpobj. Turn off compression
3402     * because sync-to-convergence takes longer if the blocksize
3403     * keeps changing.
3404     */
3405     obj = bpobj_alloc(spa->spa_meta_objset, 1 << 14, tx);
3406     dmu_object_set_compress(spa->spa_meta_objset, obj,
3407     ZIO_COMPRESS_OFF, tx);
3408     if (zap_add(spa->spa_meta_objset,
3409     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_SYNC_BPOBJ,
3410     sizeof (uint64_t), 1, &obj, tx) != 0) {
3411         cmn_err(CE_PANIC, "failed to add bpobj");
3412     }
3413     VERIFY0(bpobj_open(&spa->spa_deferred_bpobj,
3414     VERIFY3U(0, ==, bpobj_open(&spa->spa_deferred_bpobj,
3415     spa->spa_meta_objset, obj)));
3416     /*
3417     * Create the pool's history object.
3418     */
3419     if (version >= SPA_VERSION_ZPOOL_HISTORY)
3420         spa_history_create_obj(spa, tx);
3422     /*
3423     * Set pool properties.
3424     */
3425     spa->spa_bootfs = zpool_prop_default_numeric(ZPOOL_PROP_BOOTFS);
3426     spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
3427     spa->spa_failmode = zpool_prop_default_numeric(ZPOOL_PROP_FAILUREMODE);
3428     spa->spa_autoexpand = zpool_prop_default_numeric(ZPOOL_PROP_AUTOEXPAND);
3430     if (props != NULL) {
3431         spa_configfile_set(spa, props, B_FALSE);
3432         spa_sync_props(spa, props, tx);
3433     }
3435     dmu_tx_commit(tx);

```



```

5770     spa->spa_comment = spa_strdup(strval);
5771     /*
5772      * We need to dirty the configuration on all the vdevs
5773      * so that their labels get updated. It's unnecessary
5774      * to do this for pool creation since the vdev's
5775      * configuratoin has already been dirtied.
5776      */
5777     if (tx->tx_txg != TXG_INITIAL)
5778         vdev_config_dirty(spa->spa_root_vdev);
5779     spa_history_log_internal(spa, "set", tx,
5780         "%s=%s", nvpair_name(elem), strval);
5781     break;
5782 default:
5783     /*
5784      * Set pool property values in the poolprops mos object.
5785      */
5786     if (spa->spa_pool_props_object == 0) {
5787         spa->spa_pool_props_object =
5788             zap_create_link(mos, DMU_OT_POOL_PROPS,
5789                 DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_PROPS,
5790                 tx);
5791     }
5792
5793     /* normalize the property name */
5794     propname = zpool_prop_to_name(prop);
5795     proptype = zpool_prop_get_type(prop);
5796
5797     if (nvpair_type(elem) == DATA_TYPE_STRING) {
5798         ASSERT(proptype == PROP_TYPE_STRING);
5799         VERIFY(nvpair_value_string(elem, &strval) == 0);
5800         VERIFY(zap_update(mos,
5801             spa->spa_pool_props_object, propname,
5802             1, strlen(strval) + 1, strval, tx) == 0);
5803         spa_history_log_internal(spa, "set", tx,
5804             "%s=%s", nvpair_name(elem), strval);
5805     } else if (nvpair_type(elem) == DATA_TYPE_UINT64) {
5806         VERIFY(nvpair_value_uint64(elem, &intval) == 0);
5807
5808         if (proptype == PROP_TYPE_INDEX) {
5809             const char *unused;
5810             VERIFY(zpool_prop_index_to_string(
5811                 prop, intval, &unused) == 0);
5812         }
5813         VERIFY(zap_update(mos,
5814             spa->spa_pool_props_object, propname,
5815             8, 1, &intval, tx) == 0);
5816         spa_history_log_internal(spa, "set", tx,
5817             "%s=%lld", nvpair_name(elem), intval);
5818     } else {
5819         ASSERT(0); /* not allowed */
5820     }
5821
5822     switch (prop) {
5823     case ZPOOL_PROP_DELEGATION:
5824         spa->spa_delegation = intval;
5825         break;
5826     case ZPOOL_PROP_BOOTFS:
5827         spa->spa_bootfs = intval;
5828         break;
5829     case ZPOOL_PROP_FAILUREMODE:
5830         spa->spa_failmode = intval;
5831         break;
5832     case ZPOOL_PROP_AUTOEXPAND:
5833         spa->spa_autoexpand = intval;
5834         if (tx->tx_txg != TXG_INITIAL)
5835             spa_async_request(spa,

```

```

5836             SPA_ASYNC_AUTOEXPAND);
5837         break;
5838     case ZPOOL_PROP_DEDUPDITTO:
5839         spa->spa_dedup_ditto = intval;
5840         break;
5841     default:
5842         break;
5843     }
5844 }
5845
5846 }
5847
5848     mutex_exit(&spa->spa_props_lock);
5849 }
5850
5851     unchanged_portion_omitted
5852
5853     /*
5854      * Sync the specified transaction group. New blocks may be dirtied as
5855      * part of the process, so we iterate until it converges.
5856      */
5857     void
5858     spa_sync(spa_t *spa, uint64_t txg)
5859     {
5860         dsl_pool_t *dp = spa->spa_dsl_pool;
5861         objset_t *mos = spa->spa_meta_objset;
5862         bpobj_t *defer_bpo = &spa->spa_deferred_bpobj;
5863         bplist_t *free_bpl = &spa->spa_free_bplist[txg & TXG_MASK];
5864         vdev_t *rvd = spa->spa_root_vdev;
5865         vdev_t *vd;
5866         dmu_tx_t *tx;
5867         int error;
5868
5869         VERIFY(spa_writeable(spa));
5870
5871         /*
5872          * Lock out configuration changes.
5873          */
5874         spa_config_enter(spa, SCL_CONFIG, FTAG, RW_READER);
5875
5876         spa->spa_syncing_txg = txg;
5877         spa->spa_sync_pass = 0;
5878
5879         /*
5880          * If there are any pending vdev state changes, convert them
5881          * into config changes that go out with this transaction group.
5882          */
5883         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
5884         while (list_head(&spa->spa_state_dirty_list) != NULL) {
5885             /*
5886              * We need the write lock here because, for aux vdevs,
5887              * calling vdev_config_dirty() modifies sav_config.
5888              * This is ugly and will become unnecessary when we
5889              * eliminate the aux vdev wart by integrating all vdevs
5890              * into the root vdev tree.
5891              */
5892             spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
5893             spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_WRITER);
5894             while ((vd = list_head(&spa->spa_state_dirty_list)) != NULL) {
5895                 vdev_state_clean(vd);
5896                 vdev_config_dirty(vd);
5897             }
5898             spa_config_exit(spa, SCL_CONFIG | SCL_STATE, FTAG);
5899             spa_config_enter(spa, SCL_CONFIG | SCL_STATE, FTAG, RW_READER);
5900         }
5901         spa_config_exit(spa, SCL_STATE, FTAG);

```

```

5942     tx = dmu_tx_create_assigned(dp, txg);
5943
5944     /*
5945     * If we are upgrading to SPA_VERSION_RAIDZ_DEFLATE this txg,
5946     * set spa_deflate if we have no raid-z vdevs.
5947     */
5948     if (spa->spa_uberblock.ub_version < SPA_VERSION_RAIDZ_DEFLATE &&
5949         spa->spa_uberblock.ub_version >= SPA_VERSION_RAIDZ_DEFLATE) {
5950         int i;
5951
5952         for (i = 0; i < rvd->vdev_children; i++) {
5953             vdev_t *vd = rvd->vdev_child[i];
5954             if (vd->vdev_deflate_ratio != SPA_MINBLOCKSIZE)
5955                 break;
5956         }
5957         if (i == rvd->vdev_children) {
5958             spa->spa_deflate = TRUE;
5959             VERIFY0(0 == zap_add(spa->spa_meta_objset,
5960                 DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_DEFLATE,
5961                 sizeof (uint64_t), 1, &spa->spa_deflate, tx));
5962         }
5963     }
5964
5965     /*
5966     * If anything has changed in this txg, or if someone is waiting
5967     * for this txg to sync (eg, spa_vdev_remove()), push the
5968     * deferred frees from the previous txg. If not, leave them
5969     * alone so that we don't generate work on an otherwise idle
5970     * system.
5971     */
5972     if (!txg_list_empty(&dp->dp_dirty_datasets, txg) ||
5973         !txg_list_empty(&dp->dp_dirty_dirs, txg) ||
5974         !txg_list_empty(&dp->dp_sync_tasks, txg) ||
5975         ((dsl_scan_active(dp->dp_scan) ||
5976         txg_sync_waiting(dp)) && !spa_shutting_down(spa))) {
5977         zio_t *zio = zio_root(spa, NULL, NULL, 0);
5978         VERIFY3U(bpobj_iterate(defer_bpo,
5979             spa_free_sync_cb, zio, tx), ==, 0);
5980         VERIFY0(zio_wait(zio));
5981         VERIFY3U(zio_wait(zio), ==, 0);
5982     }
5983
5984     /*
5985     * Iterate to convergence.
5986     */
5987     do {
5988         int pass = ++spa->spa_sync_pass;
5989
5990         spa_sync_config_object(spa, tx);
5991         spa_sync_aux_dev(spa, &spa->spa_spare, tx,
5992             ZPOOL_CONFIG_SPARES, DMU_POOL_SPARES);
5993         spa_sync_aux_dev(spa, &spa->spa_l2cache, tx,
5994             ZPOOL_CONFIG_L2CACHE, DMU_POOL_L2CACHE);
5995         spa_errlog_sync(spa, tx);
5996         dsl_pool_sync(dp, txg);
5997
5998         if (pass <= SYNC_PASS_DEFERRED_FREE) {
5999             zio_t *zio = zio_root(spa, NULL, NULL, 0);
6000             bplist_iterate(free_bpl, spa_free_sync_cb,
6001                 zio, tx);
6002             VERIFY(zio_wait(zio) == 0);
6003         } else {
6004             bplist_iterate(free_bpl, bpobj_enqueue_cb,
6005                 defer_bpo, tx);
6006         }
6007     }

```

```

6007         ddt_sync(spa, txg);
6008         dsl_scan_sync(dp, tx);
6009
6010         while (vd = txg_list_remove(&spa->spa_vdev_txg_list, txg))
6011             vdev_sync(vd, txg);
6012
6013         if (pass == 1)
6014             spa_sync_upgrades(spa, tx);
6015
6016     } while (dmu_objset_is_dirty(mos, txg));
6017
6018     /*
6019     * Rewrite the vdev configuration (which includes the uberblock)
6020     * to commit the transaction group.
6021     *
6022     * If there are no dirty vdevs, we sync the uberblock to a few
6023     * random top-level vdevs that are known to be visible in the
6024     * config cache (see spa_vdev_add() for a complete description).
6025     * If there *are* dirty vdevs, sync the uberblock to all vdevs.
6026     */
6027     for (;;) {
6028         /*
6029         * We hold SCL_STATE to prevent vdev open/close/etc.
6030         * while we're attempting to write the vdev labels.
6031         */
6032         spa_config_enter(spa, SCL_STATE, FTAG, RW_READER);
6033
6034         if (list_is_empty(&spa->spa_config_dirty_list)) {
6035             vdev_t *svd[SPA_DVAS_PER_BP];
6036             int svdcount = 0;
6037             int children = rvd->vdev_children;
6038             int c0 = spa_get_random(children);
6039
6040             for (int c = 0; c < children; c++) {
6041                 vdev_t *vd = rvd->vdev_child[(c0 + c) % children];
6042                 if (vd->vdev_ms_array == 0 || vd->vdev_islog)
6043                     continue;
6044                 svd[svdcount++] = vd;
6045                 if (svdcount == SPA_DVAS_PER_BP)
6046                     break;
6047             }
6048             error = vdev_config_sync(svd, svdcount, txg, B_FALSE);
6049             if (error != 0)
6050                 error = vdev_config_sync(svd, svdcount, txg,
6051                     B_TRUE);
6052         } else {
6053             error = vdev_config_sync(rvd->vdev_child,
6054                 rvd->vdev_children, txg, B_FALSE);
6055             if (error != 0)
6056                 error = vdev_config_sync(rvd->vdev_child,
6057                     rvd->vdev_children, txg, B_TRUE);
6058         }
6059
6060         spa_config_exit(spa, SCL_STATE, FTAG);
6061
6062         if (error == 0)
6063             break;
6064         zio_suspend(spa, NULL);
6065         zio_resume_wait(spa);
6066     }
6067     dmu_tx_commit(tx);
6068
6069     /*
6070     * Clear the dirty config list.
6071     */
6072     while ((vd = list_head(&spa->spa_config_dirty_list)) != NULL)

```



```
6073         vdev_config_clean(vd);
6075     /*
6076      * Now that the new config has synced transactionally,
6077      * let it become visible to the config cache.
6078      */
6079     if (spa->spa_config_syncing != NULL) {
6080         spa_config_set(spa, spa->spa_config_syncing);
6081         spa->spa_config_txg = txg;
6082         spa->spa_config_syncing = NULL;
6083     }
6085     spa->spa_ubsync = spa->spa_uberblock;
6087     dsl_pool_sync_done(dp, txg);
6089     /*
6090      * Update usable space statistics.
6091      */
6092     while (vd = txg_list_remove(&spa->spa_vdev_txg_list, TXG_CLEAN(txg)))
6093         vdev_sync_done(vd, txg);
6095     spa_update_dspace(spa);
6097     /*
6098      * It had better be the case that we didn't dirty anything
6099      * since vdev_config_sync().
6100      */
6101     ASSERT(txg_list_empty(&dp->dp_dirty_datasets, txg));
6102     ASSERT(txg_list_empty(&dp->dp_dirty_dirs, txg));
6103     ASSERT(txg_list_empty(&spa->spa_vdev_txg_list, txg));
6105     spa->spa_sync_pass = 0;
6107     spa_config_exit(spa, SCL_CONFIG, FTAG);
6109     spa_handle_ignored_writes(spa);
6111     /*
6112      * If any async tasks have been requested, kick them off.
6113      */
6114     spa_async_dispatch(spa);
6115 }
_____unchanged_portion_omitted_____
```

```

*****
15272 Thu Jul 26 14:19:21 2012
new/usr/src/uts/common/fs/zfs/space_map.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 2009 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */
25 */
26
27 /*
28 * Copyright (c) 2012 by Delphix. All rights reserved.
29 */
30
31 #include <sys/zfs_context.h>
32 #include <sys/spa.h>
33 #include <sys/dmu.h>
34 #include <sys/zio.h>
35 #include <sys/space_map.h>
36
37 /*
38  * Space map routines.
39  * NOTE: caller is responsible for all locking.
40  */
41 static int
42 space_map_seg_compare(const void *x1, const void *x2)
43 {
44     const space_seg_t *s1 = x1;
45     const space_seg_t *s2 = x2;
46
47     if (s1->ss_start < s2->ss_start) {
48         if (s1->ss_end > s2->ss_start)
49             return (0);
50         return (-1);
51     }
52     if (s1->ss_start > s2->ss_start) {
53         if (s1->ss_start < s2->ss_end)
54             return (0);
55         return (1);
56     }
57     return (0);
58 }
59
60 unchanged portion omitted
61
62 void

```

```

78 space_map_destroy(space_map_t *sm)
79 {
80     ASSERT(!sm->sm_loaded && !sm->sm_loading);
81     VERIFY0(sm->sm_space);
82     VERIFY3U(sm->sm_space, ==, 0);
83     avl_destroy(&sm->sm_root);
84     cv_destroy(&sm->sm_load_cv);
85 }
86
87 unchanged portion omitted
88
89 /*
90  * Note: space_map_load() will drop sm_lock across dmu_read() calls.
91  * The caller must be OK with this.
92  */
93 int
94 space_map_load(space_map_t *sm, space_map_ops_t *ops, uint8_t maptype,
95               space_map_obj_t *smo, objset_t *os)
96 {
97     uint64_t *entry, *entry_map, *entry_map_end;
98     uint64_t bufsize, size, offset, end, space;
99     uint64_t mapstart = sm->sm_start;
100    int error = 0;
101
102    ASSERT(MUTEX_HELD(sm->sm_lock));
103    ASSERT(!sm->sm_loaded);
104    ASSERT(!sm->sm_loading);
105
106    sm->sm_loading = B_TRUE;
107    end = smo->smo_objsize;
108    space = smo->smo_alloc;
109
110    ASSERT(sm->sm_ops == NULL);
111    VERIFY0(sm->sm_space);
112    VERIFY3U(sm->sm_space, ==, 0);
113
114    if (maptype == SM_FREE) {
115        space_map_add(sm, sm->sm_start, sm->sm_size);
116        space = sm->sm_size - space;
117    }
118
119    bufsize = 1ULL << SPACE_MAP_BLOCKSHIFT;
120    entry_map = zio_buf_alloc(bufsize);
121
122    mutex_exit(sm->sm_lock);
123    if (end > bufsize)
124        dmu_prefetch(os, smo->smo_object, bufsize, end - bufsize);
125    mutex_enter(sm->sm_lock);
126
127    for (offset = 0; offset < end; offset += bufsize) {
128        size = MIN(end - offset, bufsize);
129        VERIFY(P2PHASE(size, sizeof (uint64_t)) == 0);
130        VERIFY(size != 0);
131
132        dprintf("object=%llu offset=%llx size=%llx\n",
133              smo->smo_object, offset, size);
134
135        mutex_exit(sm->sm_lock);
136        error = dmu_read(os, smo->smo_object, offset, size, entry_map,
137                      DMU_READ_PREFETCH);
138        mutex_enter(sm->sm_lock);
139        if (error != 0)
140            break;
141
142        entry_map_end = entry_map + (size / sizeof (uint64_t));
143        for (entry = entry_map; entry < entry_map_end; entry++) {
144            uint64_t e = *entry;

```

```

328             if (SM_DEBUG_DECODE(e))           /* Skip debug entries */
329                 continue;

331             (SM_TYPE_DECODE(e) == matype ?
332              space_map_add : space_map_remove)(sm,
333              (SM_OFFSET_DECODE(e) << sm->sm_shift) + mapstart,
334              SM_RUN_DECODE(e) << sm->sm_shift);
335         }
336     }

338     if (error == 0) {
339         VERIFY3U(sm->sm_space, ==, space);

341         sm->sm_loaded = B_TRUE;
342         sm->sm_ops = ops;
343         if (ops != NULL)
344             ops->smop_load(sm);
345     } else {
346         space_map_vacate(sm, NULL, NULL);
347     }

349     zio_buf_free(entry_map, bufsize);

351     sm->sm_loading = B_FALSE;

353     cv_broadcast(&sm->sm_load_cv);

355     return (error);
356 }

```

unchanged portion omitted

```

404 /*
405  * Note: space_map_sync() will drop sm_lock across dmu_write() calls.
406  */
407 void
408 space_map_sync(space_map_t *sm, uint8_t matype,
409               space_map_obj_t *smo, objset_t *os, dmu_tx_t *tx)
410 {
411     spa_t *spa = dmu_objset_spa(os);
412     void *cookie = NULL;
413     space_seg_t *ss;
414     uint64_t bufsize, start, size, run_len;
415     uint64_t *entry, *entry_map, *entry_map_end;

417     ASSERT(MUTEX_HELD(sm->sm_lock));

419     if (sm->sm_space == 0)
420         return;

422     dprintf("object %llu, txg %llu, pass %d, %c, count %lu, space %llx\n",
423            smo->smo_object, dmu_tx_get_txg(tx), spa_sync_pass(spa),
424            matype == SM_ALLOC ? 'A' : 'F', avl_numnodes(&sm->sm_root),
425            sm->sm_space);

427     if (matype == SM_ALLOC)
428         smo->smo_alloc += sm->sm_space;
429     else
430         smo->smo_alloc -= sm->sm_space;

432     bufsize = (8 + avl_numnodes(&sm->sm_root)) * sizeof (uint64_t);
433     bufsize = MIN(bufsize, 1ULL << SPACE_MAP_BLOCKSHIFT);
434     entry_map = zio_buf_alloc(bufsize);
435     entry_map_end = entry_map + (bufsize / sizeof (uint64_t));
436     entry = entry_map;

```

```

438     *entry++ = SM_DEBUG_ENCODE(1) |
439              SM_DEBUG_ACTION_ENCODE(matype) |
440              SM_DEBUG_SYNCPASS_ENCODE(spa_sync_pass(spa)) |
441              SM_DEBUG_TXG_ENCODE(dmu_tx_get_txg(tx));

443     while ((ss = avl_destroy_nodes(&sm->sm_root, &cookie)) != NULL) {
444         size = ss->ss_end - ss->ss_start;
445         start = (ss->ss_start - sm->sm_start) >> sm->sm_shift;

447         sm->sm_space -= size;
448         size >>= sm->sm_shift;

450         while (size) {
451             run_len = MIN(size, SM_RUN_MAX);

453             if (entry == entry_map_end) {
454                 mutex_exit(sm->sm_lock);
455                 dmu_write(os, smo->smo_object, smo->smo_objsize,
456                          bufsize, entry_map, tx);
457                 mutex_enter(sm->sm_lock);
458                 smo->smo_objsize += bufsize;
459                 entry = entry_map;
460             }

462             *entry++ = SM_OFFSET_ENCODE(start) |
463                      SM_TYPE_ENCODE(matype) |
464                      SM_RUN_ENCODE(run_len);

466             start += run_len;
467             size -= run_len;
468         }
469         kmem_free(ss, sizeof (*ss));
470     }

472     if (entry != entry_map) {
473         size = (entry - entry_map) * sizeof (uint64_t);
474         mutex_exit(sm->sm_lock);
475         dmu_write(os, smo->smo_object, smo->smo_objsize,
476                  size, entry_map, tx);
477         mutex_enter(sm->sm_lock);
478         smo->smo_objsize += size;
479     }

481     zio_buf_free(entry_map, bufsize);

483     VERIFY0(sm->sm_space);
478     VERIFY3U(sm->sm_space, ==, 0);
484 }

```

unchanged portion omitted

```

*****
85047 Thu Jul 26 14:19:22 2012
new/usr/src/uts/common/fs/zfs/vdev.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

564 void
565 vdev_free(vdev_t *vd)
566 {
567     spa_t *spa = vd->vdev_spa;

569     /*
570      * vdev_free() implies closing the vdev first. This is simpler than
571      * trying to ensure complicated semantics for all callers.
572      */
573     vdev_close(vd);

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

578     /*
579      * Free all children.
580      */
581     for (int c = 0; c < vd->vdev_children; c++)
582         vdev_free(vd->vdev_child[c]);

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

587     /*
588      * Discard allocation state.
589      */
590     if (vd->vdev_mg != NULL) {
591         vdev metaslab_fini(vd);
592         metaslab_group_destroy(vd->vdev_mg);
593     }

595     ASSERT0(vd->vdev_stat.vs_space);
596     ASSERT0(vd->vdev_stat.vs_dspace);
597     ASSERT0(vd->vdev_stat.vs_alloc);
598     ASSERT3U(vd->vdev_stat.vs_space, ==, 0);
599     ASSERT3U(vd->vdev_stat.vs_dspace, ==, 0);
600     ASSERT3U(vd->vdev_stat.vs_alloc, ==, 0);

602     /*
603      * Remove this vdev from its parent's child list.
604      */
605     vdev_remove_child(vd->vdev_parent, vd);

607     ASSERT(vd->vdev_parent == NULL);

609     /*
610      * Clean up vdev structure.
611      */
612     vdev_queue_fini(vd);
613     vdev_cache_fini(vd);

615     if (vd->vdev_path)
616         spa_strfree(vd->vdev_path);
617     if (vd->vdev_devid)
618         spa_strfree(vd->vdev_devid);
619     if (vd->vdev_physpath)
620         spa_strfree(vd->vdev_physpath);
621     if (vd->vdev_fru)
622         spa_strfree(vd->vdev_fru);
623     if (vd->vdev_fru)
624         spa_strfree(vd->vdev_fru);

```

```

621     if (vd->vdev_isspare)
622         spa_spare_remove(vd);
623     if (vd->vdev_isl2cache)
624         spa_l2cache_remove(vd);

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

629     mutex_enter(&vd->vdev_dtl_lock);
630     for (int t = 0; t < DTL_TYPES; t++) {
631         space_map_unload(&vd->vdev_dtl[t]);
632         space_map_destroy(&vd->vdev_dtl[t]);
633     }
634     mutex_exit(&vd->vdev_dtl_lock);

636     mutex_destroy(&vd->vdev_dtl_lock);
637     mutex_destroy(&vd->vdev_stat_lock);
638     mutex_destroy(&vd->vdev_probe_lock);

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

643     kmem_free(vd, sizeof (vdev_t));
644 }
_____unchanged_portion_omitted_____

1789 void
1790 vdev_dtl_sync(vdev_t *vd, uint64_t txg)
1791 {
1792     spa_t *spa = vd->vdev_spa;
1793     space_map_obj_t *smo = &vd->vdev_dtl_smo;
1794     space_map_t *sm = &vd->vdev_dtl[DTL_MISSING];
1795     objset_t *mos = spa->spa_meta_objset;
1796     space_map_t smsync;
1797     kmutex_t smlock;
1798     dmu_buf_t *db;
1799     dmu_tx_t *tx;

1801     ASSERT(!vd->vdev_ishole);

1803     tx = dmu_tx_create_assigned(spa->spa_dsl_pool, txg);

1805     if (vd->vdev_detached) {
1806         if (smo->smo_object != 0) {
1807             int err = dmu_object_free(mos, smo->smo_object, tx);
1808             ASSERT0(err);
1809             ASSERT3U(err, ==, 0);
1810             smo->smo_object = 0;
1811         }
1812         dmu_tx_commit(tx);
1813         return;
1814     }

1815     if (smo->smo_object == 0) {
1816         ASSERT(smo->smo_objsize == 0);
1817         ASSERT(smo->smo_alloc == 0);
1818         smo->smo_object = dmu_object_alloc(mos,
1819             DMU_OT_SPACE_MAP, 1 << SPACE_MAP_BLOCKSHIFT,
1820             DMU_OT_SPACE_MAP_HEADER, sizeof (*smo), tx);
1821         ASSERT(smo->smo_object != 0);
1822         vdev_config_dirty(vd->vdev_top);
1823     }

1825     mutex_init(&smlock, NULL, MUTEX_DEFAULT, NULL);

```

```

1827     space_map_create(&smsync, sm->sm_start, sm->sm_size, sm->sm_shift,
1828                     &smlock);
1830     mutex_enter(&smlock);
1832     mutex_enter(&vd->vdev_dtl_lock);
1833     space_map_walk(sm, space_map_add, &smsync);
1834     mutex_exit(&vd->vdev_dtl_lock);
1836     space_map_truncate(smo, mos, tx);
1837     space_map_sync(&smsync, SM_ALLOC, smo, mos, tx);
1839     space_map_destroy(&smsync);
1841     mutex_exit(&smlock);
1842     mutex_destroy(&smlock);
1844     VERIFY(0 == dmu_bonus_hold(mos, smo->smo_object, FTAG, &db));
1845     dmu_buf_will_dirty(db, tx);
1846     ASSERT3U(db->db_size, >=, sizeof (*smo));
1847     bcopy(smo, db->db_data, sizeof (*smo));
1848     dmu_buf_rele(db, FTAG);
1850     dmu_tx_commit(tx);
1851 }
    unchanged portion omitted
1998 void
1999 vdev_remove(vdev_t *vd, uint64_t txg)
2000 {
2001     spa_t *spa = vd->vdev_spa;
2002     objset_t *mos = spa->spa_meta_objset;
2003     dmu_tx_t *tx;
2005     tx = dmu_tx_create_assigned(spa_get_dsl(spa), txg);
2007     if (vd->vdev_dtl_smo.smo_object) {
2008         ASSERT0(vd->vdev_dtl_smo.smo_alloc);
2008         ASSERT3U(vd->vdev_dtl_smo.smo_alloc, ==, 0);
2009         (void) dmu_object_free(mos, vd->vdev_dtl_smo.smo_object, tx);
2010         vd->vdev_dtl_smo.smo_object = 0;
2011     }
2013     if (vd->vdev_ms != NULL) {
2014         for (int m = 0; m < vd->vdev_ms_count; m++) {
2015             metaslab_t *msp = vd->vdev_ms[m];
2017             if (msp == NULL || msp->ms_smo.smo_object == 0)
2018                 continue;
2020             ASSERT0(msp->ms_smo.smo_alloc);
2020             ASSERT3U(msp->ms_smo.smo_alloc, ==, 0);
2021             (void) dmu_object_free(mos, msp->ms_smo.smo_object, tx);
2022             msp->ms_smo.smo_object = 0;
2023         }
2024     }
2026     if (vd->vdev_ms_array) {
2027         (void) dmu_object_free(mos, vd->vdev_ms_array, tx);
2028         vd->vdev_ms_array = 0;
2029         vd->vdev_ms_shift = 0;
2030     }
2031     dmu_tx_commit(tx);
2032 }
    unchanged portion omitted

```

```

2236 static int
2237 vdev_offline_locked(spa_t *spa, uint64_t guid, uint64_t flags)
2238 {
2239     vdev_t *vd, *tvd;
2240     int error = 0;
2241     uint64_t generation;
2242     metaslab_group_t *mg;
2244 top:
2245     spa_vdev_state_enter(spa, SCL_ALLOC);
2247     if ((vd = spa_lookup_by_guid(spa, guid, B_TRUE)) == NULL)
2248         return (spa_vdev_state_exit(spa, NULL, ENODEV));
2250     if (!vd->vdev_ops->vdev_op_leaf)
2251         return (spa_vdev_state_exit(spa, NULL, ENOTSUP));
2253     tvd = vd->vdev_top;
2254     mg = tvd->vdev_mg;
2255     generation = spa->spa_config_generation + 1;
2257     /*
2258      * If the device isn't already offline, try to offline it.
2259      */
2260     if (!vd->vdev_offline) {
2261         /*
2262          * If this device has the only valid copy of some data,
2263          * don't allow it to be offlined. Log devices are always
2264          * expendable.
2265          */
2266         if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2267             vdev_dtl_required(vd))
2268             return (spa_vdev_state_exit(spa, NULL, EBUSY));
2270         /*
2271          * If the top-level is a slog and it has had allocations
2272          * then proceed. We check that the vdev's metaslab group
2273          * is not NULL since it's possible that we may have just
2274          * added this vdev but not yet initialized its metaslabs.
2275          */
2276         if (tvd->vdev_islog && mg != NULL) {
2277             /*
2278              * Prevent any future allocations.
2279              */
2280             metaslab_group_passivate(mg);
2281             (void) spa_vdev_state_exit(spa, vd, 0);
2283             error = spa_offline_log(spa);
2285             spa_vdev_state_enter(spa, SCL_ALLOC);
2287             /*
2288              * Check to see if the config has changed.
2289              */
2290             if (error || generation != spa->spa_config_generation) {
2291                 metaslab_group_activate(mg);
2292                 if (error)
2293                     return (spa_vdev_state_exit(spa,
2294                                                 vd, error));
2295                 (void) spa_vdev_state_exit(spa, vd, 0);
2296                 goto top;
2297             }
2298             ASSERT0(tvd->vdev_stat.vs_alloc);
2298             ASSERT3U(tvd->vdev_stat.vs_alloc, ==, 0);
2299         }

```

```
2301     /*
2302     * Offline this device and reopen its top-level vdev.
2303     * If the top-level vdev is a log device then just offline
2304     * it. Otherwise, if this action results in the top-level
2305     * vdev becoming unusable, undo it and fail the request.
2306     */
2307     vd->vdev_offline = B_TRUE;
2308     vdev_reopen(tvd);
2310     if (!tvd->vdev_islog && vd->vdev_aux == NULL &&
2311         vdev_is_dead(tvd)) {
2312         vd->vdev_offline = B_FALSE;
2313         vdev_reopen(tvd);
2314         return (spa_vdev_state_exit(spa, NULL, EBUSY));
2315     }
2317     /*
2318     * Add the device back into the metaslab rotor so that
2319     * once we online the device it's open for business.
2320     */
2321     if (tvd->vdev_islog && mg != NULL)
2322         metaslab_group_activate(mg);
2323 }
2325     vd->vdev_tmpoffline = !(flags & ZFS_OFFLINE_TEMPORARY);
2327     return (spa_vdev_state_exit(spa, vd, 0));
2328 }
unchanged_portion_omitted
```

```

*****
61791 Thu Jul 26 14:19:23 2012
new/usr/src/uts/common/fs/zfs/vdev_raidz.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

279 static void
280 vdev_raidz_map_free_vsd(zio_t *zio)
281 {
282     raidz_map_t *rm = zio->io_vsd;

284     ASSERT0(rm->rm_freed);
284     ASSERT3U(rm->rm_freed, ==, 0);
285     rm->rm_freed = 1;

287     if (rm->rm_reports == 0)
288         vdev_raidz_map_free(rm);
289 }
_____unchanged_portion_omitted_____

1090 static void
1091 vdev_raidz_matrix_invert(raidz_map_t *rm, int n, int nmissing, int *missing,
1092     uint8_t **rows, uint8_t **invrows, const uint8_t *used)
1093 {
1094     int i, j, ii, jj;
1095     uint8_t log;

1097     /*
1098     * Assert that the first nmissing entries from the array of used
1099     * columns correspond to parity columns and that subsequent entries
1100     * correspond to data columns.
1101     */
1102     for (i = 0; i < nmissing; i++) {
1103         ASSERT3S(used[i], <, rm->rm_firstdatacol);
1104     }
1105     for (; i < n; i++) {
1106         ASSERT3S(used[i], >=, rm->rm_firstdatacol);
1107     }

1109     /*
1110     * First initialize the storage where we'll compute the inverse rows.
1111     */
1112     for (i = 0; i < nmissing; i++) {
1113         for (j = 0; j < n; j++) {
1114             invrows[i][j] = (i == j) ? 1 : 0;
1115         }
1116     }

1118     /*
1119     * Subtract all trivial rows from the rows of consequence.
1120     */
1121     for (i = 0; i < nmissing; i++) {
1122         for (j = nmissing; j < n; j++) {
1123             ASSERT3U(used[j], >=, rm->rm_firstdatacol);
1124             jj = used[j] - rm->rm_firstdatacol;
1125             ASSERT3S(jj, <, n);
1126             invrows[i][j] = rows[i][jj];
1127             rows[i][jj] = 0;
1128         }
1129     }

1131     /*
1132     * For each of the rows of interest, we must normalize it and subtract
1133     * a multiple of it from the other rows.
1134     */

```

```

1135     for (i = 0; i < nmissing; i++) {
1136         for (j = 0; j < missing[i]; j++) {
1137             ASSERT0(rows[i][j]);
1137             ASSERT3U(rows[i][j], ==, 0);
1138         }
1139         ASSERT3U(rows[i][missing[i]], !=, 0);

1141         /*
1142         * Compute the inverse of the first element and multiply each
1143         * element in the row by that value.
1144         */
1145         log = 255 - vdev_raidz_log2[rows[i][missing[i]]];

1147         for (j = 0; j < n; j++) {
1148             rows[i][j] = vdev_raidz_exp2(rows[i][j], log);
1149             invrows[i][j] = vdev_raidz_exp2(invrows[i][j], log);
1150         }

1152         for (ii = 0; ii < nmissing; ii++) {
1153             if (i == ii)
1154                 continue;

1156             ASSERT3U(rows[ii][missing[i]], !=, 0);

1158             log = vdev_raidz_log2[rows[ii][missing[i]]];

1160             for (j = 0; j < n; j++) {
1161                 rows[ii][j] ^=
1162                     vdev_raidz_exp2(rows[i][j], log);
1163                 invrows[ii][j] ^=
1164                     vdev_raidz_exp2(invrows[i][j], log);
1165             }
1166         }
1167     }

1169     /*
1170     * Verify that the data that is left in the rows are properly part of
1171     * an identity matrix.
1172     */
1173     for (i = 0; i < nmissing; i++) {
1174         for (j = 0; j < n; j++) {
1175             if (j == missing[i]) {
1176                 ASSERT3U(rows[i][j], ==, 1);
1177             } else {
1178                 ASSERT0(rows[i][j]);
1178                 ASSERT3U(rows[i][j], ==, 0);
1179             }
1180         }
1181     }
1182 }
_____unchanged_portion_omitted_____

```

```

*****
33603 Thu Jul 26 14:19:24 2012
new/usr/src/uts/common/fs/zfs/zap.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

140 /*
141  * Generic routines for dealing with the pointer & cookie tables.
142  */

144 static int
145 zap_table_grow(zap_t *zap, zap_table_phys_t *tbl,
146               void (*transfer_func)(const uint64_t *src, uint64_t *dst, int n),
147               dmu_tx_t *tx)
148 {
149     uint64_t b, newblk;
150     dmu_buf_t *db_old, *db_new;
151     int err;
152     int bs = FZAP_BLOCK_SHIFT(zap);
153     int hepb = 1<<(bs-4);
154     /* hepb = half the number of entries in a block */

156     ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
157     ASSERT(tbl->zt_blk != 0);
158     ASSERT(tbl->zt_numblks > 0);

160     if (tbl->zt_nextblk != 0) {
161         newblk = tbl->zt_nextblk;
162     } else {
163         newblk = zap_allocate_blocks(zap, tbl->zt_numblks * 2);
164         tbl->zt_nextblk = newblk;
165         ASSERT0(tbl->zt_blks_copied);
166         ASSERT3U(tbl->zt_blks_copied, ==, 0);
167         dmu_prefetch(zap->zap_objset, zap->zap_object,
168                   tbl->zt_blk << bs, tbl->zt_numblks << bs);
169     }

170     /*
171      * Copy the ptrtbl from the old to new location.
172      */

174     b = tbl->zt_blks_copied;
175     err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
176                   (tbl->zt_blk + b) << bs, FTAG, &db_old, DMU_READ_NO_PREFETCH);
177     if (err)
178         return (err);

180     /* first half of entries in old[b] go to new[2*b+0] */
181     VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
182                   (newblk + 2*b+0) << bs, FTAG, &db_new, DMU_READ_NO_PREFETCH));
183     dmu_buf_will_dirty(db_new, tx);
184     transfer_func(db_old->db_data, db_new->db_data, hepb);
185     dmu_buf_rele(db_new, FTAG);

187     /* second half of entries in old[b] go to new[2*b+1] */
188     VERIFY(0 == dmu_buf_hold(zap->zap_objset, zap->zap_object,
189                   (newblk + 2*b+1) << bs, FTAG, &db_new, DMU_READ_NO_PREFETCH));
190     dmu_buf_will_dirty(db_new, tx);
191     transfer_func((uint64_t *)db_old->db_data + hepb,
192                   db_new->db_data, hepb);
193     dmu_buf_rele(db_new, FTAG);

195     dmu_buf_rele(db_old, FTAG);

197     tbl->zt_blks_copied++;

```

```

199     dprintf("copied block %llu of %llu\n",
200           tbl->zt_blks_copied, tbl->zt_numblks);

202     if (tbl->zt_blks_copied == tbl->zt_numblks) {
203         (void) dmu_free_range(zap->zap_objset, zap->zap_object,
204                             tbl->zt_blk << bs, tbl->zt_numblks << bs, tx);

206         tbl->zt_blk = newblk;
207         tbl->zt_numblks *= 2;
208         tbl->zt_shift++;
209         tbl->zt_nextblk = 0;
210         tbl->zt_blks_copied = 0;

212         dprintf("finished; numblocks now %llu (%lluk entries)\n",
213               tbl->zt_numblks, 1<<(tbl->zt_shift-10));
214     }

216     return (0);
217 }
_____unchanged_portion_omitted_____

318 static int
319 zap_grow_ptrtbl(zap_t *zap, dmu_tx_t *tx)
320 {
321     /*
322      * The pointer table should never use more hash bits than we
323      * have (otherwise we'd be using useless zero bits to index it).
324      * If we are within 2 bits of running out, stop growing, since
325      * this is already an aberrant condition.
326      */
327     if (zap->zap_f.zap_phys->zap_ptrtbl.zt_shift >= zap_hashbits(zap) - 2)
328         return (ENOSPC);

330     if (zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks == 0) {
331         /*
332          * We are outgrowing the "embedded" ptrtbl (the one
333          * stored in the header block). Give it its own entire
334          * block, which will double the size of the ptrtbl.
335          */
336         uint64_t newblk;
337         dmu_buf_t *db_new;
338         int err;

340         ASSERT3U(zap->zap_f.zap_phys->zap_ptrtbl.zt_shift, ==,
341                 ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
342         ASSERT0(zap->zap_f.zap_phys->zap_ptrtbl.zt_blk);
343         ASSERT3U(zap->zap_f.zap_phys->zap_ptrtbl.zt_blk, ==, 0);

344         newblk = zap_allocate_blocks(zap, 1);
345         err = dmu_buf_hold(zap->zap_objset, zap->zap_object,
346                   newblk << FZAP_BLOCK_SHIFT(zap), FTAG, &db_new,
347                   DMU_READ_NO_PREFETCH);
348         if (err)
349             return (err);
350         dmu_buf_will_dirty(db_new, tx);
351         zap_ptrtbl_transfer(&ZAP_EMBEDDED_PTRTBL_ENT(zap, 0),
352                           db_new->db_data, 1 << ZAP_EMBEDDED_PTRTBL_SHIFT(zap));
353         dmu_buf_rele(db_new, FTAG);

355         zap->zap_f.zap_phys->zap_ptrtbl.zt_blk = newblk;
356         zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks = 1;
357         zap->zap_f.zap_phys->zap_ptrtbl.zt_shift++;

359         ASSERT3U(1ULL << zap->zap_f.zap_phys->zap_ptrtbl.zt_shift, ==,
360               zap->zap_f.zap_phys->zap_ptrtbl.zt_numblks <<

```



```

361         (FZAP_BLOCK_SHIFT(zap)-3));
363         return (0);
364     } else {
365         return (zap_table_grow(zap, &zap->zap_f.zap_phys->zap_ptrtbl,
366             zap_ptrtbl_transfer, tx));
367     }
368 }
    unchanged_portion_omitted
449 static zap_leaf_t *
450 zap_open_leaf(uint64_t blkid, dmu_buf_t *db)
451 {
452     zap_leaf_t *l, *winner;
453
454     ASSERT(blkid != 0);
455
456     l = kmem_alloc(sizeof (zap_leaf_t), KM_SLEEP);
457     rw_init(&l->l_rwlock, 0, 0, 0);
458     rw_enter(&l->l_rwlock, RW_WRITER);
459     l->l_blkid = blkid;
460     l->l_bs = highbit(db->db_size)-1;
461     l->l_dbuf = db;
462     l->l_phys = NULL;
463
464     winner = dmu_buf_set_user(db, l, &l->l_phys, zap_leaf_pageout);
465
466     rw_exit(&l->l_rwlock);
467     if (winner != NULL) {
468         /* someone else set it first */
469         zap_leaf_pageout(NULL, l);
470         l = winner;
471     }
472
473     /*
474      * lhr_pad was previously used for the next leaf in the leaf
475      * chain. There should be no chained leafs (as we have removed
476      * support for them).
477      */
478     ASSERT0(l->l_phys->l_hdr.lh_pad1);
479     ASSERT3U(l->l_phys->l_hdr.lh_pad1, ==, 0);
480
481     /*
482      * There should be more hash entries than there can be
483      * chunks to put in the hash table
484      */
485     ASSERT3U(ZAP_LEAF_HASH_NUMENTRIES(l), >, ZAP_LEAF_NUMCHUNKS(l) / 3);
486
487     /* The chunks should begin at the end of the hash table */
488     ASSERT3P(&ZAP_LEAF_CHUNK(l, 0), ==,
489         &l->l_phys->l_hash[ZAP_LEAF_HASH_NUMENTRIES(l)]);
490
491     /* The chunks should end at the end of the block */
492     ASSERT3U((uintptr_t)&ZAP_LEAF_CHUNK(l, ZAP_LEAF_NUMCHUNKS(l)) -
493         (uintptr_t)l->l_phys, ==, l->l_dbuf->db_size);
494
495     return (l);
496 }
    unchanged_portion_omitted
591 static int
592 zap_expand_leaf(zap_name_t *zn, zap_leaf_t *l, dmu_tx_t *tx, zap_leaf_t **lp)
593 {
594     zap_t *zap = zn->zn_zap;
595     uint64_t hash = zn->zn_hash;
596     zap_leaf_t *nl;

```

```

597     int prefix_diff, i, err;
598     uint64_t sibling;
599     int old_prefix_len = l->l_phys->l_hdr.lh_prefix_len;
600
601     ASSERT3U(old_prefix_len, <=, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
602     ASSERT(RW_LOCK_HELD(&zap->zap_rwlock));
603
604     ASSERT3U(ZAP_HASH_IDX(hash, old_prefix_len), ==,
605         l->l_phys->l_hdr.lh_prefix);
606
607     if (zap_tryupgradedir(zap, tx) == 0 ||
608         old_prefix_len == zap->zap_f.zap_phys->zap_ptrtbl.zt_shift) {
609         /* We failed to upgrade, or need to grow the pointer table */
610         objset_t *os = zap->zap_objset;
611         uint64_t object = zap->zap_object;
612
613         zap_put_leaf(l);
614         zap_unlockdir(zap);
615         err = zap_lockdir(os, object, tx, RW_WRITER,
616             FALSE, FALSE, &zn->zn_zap);
617         zap = zn->zn_zap;
618         if (err)
619             return (err);
620         ASSERT(!zap->zap_ismicro);
621
622         while (old_prefix_len ==
623             zap->zap_f.zap_phys->zap_ptrtbl.zt_shift) {
624             err = zap_grow_ptrtbl(zap, tx);
625             if (err)
626                 return (err);
627         }
628
629         err = zap_deref_leaf(zap, hash, tx, RW_WRITER, &l);
630         if (err)
631             return (err);
632
633         if (l->l_phys->l_hdr.lh_prefix_len != old_prefix_len) {
634             /* it split while our locks were down */
635             *lp = l;
636             return (0);
637         }
638     }
639     ASSERT(RW_WRITE_HELD(&zap->zap_rwlock));
640     ASSERT3U(old_prefix_len, <, zap->zap_f.zap_phys->zap_ptrtbl.zt_shift);
641     ASSERT3U(ZAP_HASH_IDX(hash, old_prefix_len), ==,
642         l->l_phys->l_hdr.lh_prefix);
643
644     prefix_diff = zap->zap_f.zap_phys->zap_ptrtbl.zt_shift -
645         (old_prefix_len + 1);
646     sibling = (ZAP_HASH_IDX(hash, old_prefix_len + 1) | 1) << prefix_diff;
647
648     /* check for i/o errors before doing zap_leaf_split */
649     for (i = 0; i < (1ULL<<prefix_diff); i++) {
650         uint64_t blk;
651         err = zap_idx_to_blk(zap, sibling+i, &blk);
652         if (err)
653             return (err);
654         ASSERT3U(blk, ==, l->l_blkid);
655     }
656
657     nl = zap_create_leaf(zap, tx);
658     zap_leaf_split(l, nl, zap->zap_normflags != 0);
659
660     /* set sibling pointers */
661     for (i = 0; i < (1ULL<<prefix_diff); i++) {
662         for (i = 0; i < (1ULL<<prefix_diff); i++) {

```

```
662     err = zap_set_idx_to_blk(zap, sibling+i, nl->l_blkid, tx);
663     ASSERT0(err); /* we checked for i/o errors above */
663     ASSERT3U(err, ==, 0); /* we checked for i/o errors above */
664 }

666     if (hash & (1ULL << (64 - l->l_phys->l_hdr.lh_prefix_len))) {
667         /* we want the sibling */
668         zap_put_leaf(l);
669         *lp = nl;
670     } else {
671         zap_put_leaf(nl);
672         *lp = l;
673     }

675     return (0);
676 }
unchanged_portion_omitted
```

```

*****
34759 Thu Jul 26 14:19:25 2012
new/usr/src/uts/common/fs/zfs/zap_micro.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

445 int
446 zap_lockdir(objset_t *os, uint64_t obj, dmu_tx_t *tx,
447     krw_t lti, boolean_t fatreader, boolean_t adding, zap_t **zapp)
448 {
449     zap_t *zap;
450     dmu_buf_t *db;
451     krw_t lt;
452     int err;

454     *zapp = NULL;

456     err = dmu_buf_hold(os, obj, 0, NULL, &db, DMU_READ_NO_PREFETCH);
457     if (err)
458         return (err);

460 #ifdef ZFS_DEBUG
461     {
462         dmu_object_info_t doi;
463         dmu_object_info_from_db(db, &doi);
464         ASSERT3U(DMU_OT_BYTESWAP(doi.doi_type), ==, DMU_BSWAP_ZAP);
465     }
466 #endif

468     zap = dmu_buf_get_user(db);
469     if (zap == NULL)
470         zap = mzap_open(os, obj, db);

472     /*
473     * We're checking zap_ismicro without the lock held, in order to
474     * tell what type of lock we want.  Once we have some sort of
475     * lock, see if it really is the right type.  In practice this
476     * can only be different if it was upgraded from micro to fat,
477     * and micro wanted WRITER but fat only needs READER.
478     */
479     lt = (!zap->zap_ismicro && fatreader) ? RW_READER : lti;
480     rw_enter(&zap->zap_rwlock, lt);
481     if (lt != ((!zap->zap_ismicro && fatreader) ? RW_READER : lti)) {
482         /* it was upgraded, now we only need reader */
483         ASSERT(lt == RW_WRITER);
484         ASSERT(RW_READER ==
485             (!zap->zap_ismicro && fatreader) ? RW_READER : lti);
486         rw_downgrade(&zap->zap_rwlock);
487         lt = RW_READER;
488     }

490     zap->zap_objset = os;

492     if (lt == RW_WRITER)
493         dmu_buf_will_dirty(db, tx);

495     ASSERT3P(zap->zap_dbuf, ==, db);

497     ASSERT(!zap->zap_ismicro ||
498         zap->zap_m.zap_num_entries <= zap->zap_m.zap_num_chunks);
499     if (zap->zap_ismicro && tx && adding &&
500         zap->zap_m.zap_num_entries == zap->zap_m.zap_num_chunks) {
501         uint64_t newsz = db->db_size + SPA_MINBLOCKSIZE;
502         if (newsz > MZAP_MAX_BLKSZ) {
503             dprintf("upgrading obj %llu: num_entries=%u\n",

```

```

504         obj, zap->zap_m.zap_num_entries);
505         *zapp = zap;
506         return (mzap_upgrade(zapp, tx, 0));
507     }
508     err = dmu_object_set_blocksize(os, obj, newsz, 0, tx);
509     ASSERT0(err);
509     ASSERT3U(err, ==, 0);
510     zap->zap_m.zap_num_chunks =
511         db->db_size / MZAP_ENT_LEN - 1;
512     }

514     *zapp = zap;
515     return (0);
516 }
_____unchanged_portion_omitted_____

575 static void
576 mzap_create_impl(objset_t *os, uint64_t obj, int normflags, zap_flags_t flags,
577     dmu_tx_t *tx)
578 {
579     dmu_buf_t *db;
580     mzap_phys_t *zp;

582     VERIFY(0 == dmu_buf_hold(os, obj, 0, FTAG, &db, DMU_READ_NO_PREFETCH));

584 #ifdef ZFS_DEBUG
585     {
586         dmu_object_info_t doi;
587         dmu_object_info_from_db(db, &doi);
588         ASSERT3U(DMU_OT_BYTESWAP(doi.doi_type), ==, DMU_BSWAP_ZAP);
589     }
590 #endif

592     dmu_buf_will_dirty(db, tx);
593     zp = db->db_data;
594     zp->mz_block_type = ZBT_MICRO;
595     zp->mz_salt = ((uintptr_t)db ^ (uintptr_t)tx ^ (obj << 1)) | 1ULL;
596     zp->mz_normflags = normflags;
597     dmu_buf_rele(db, FTAG);

599     if (flags != 0) {
600         zap_t *zap;
601         /* Only fat zap supports flags; upgrade immediately. */
602         VERIFY(0 == zap_lockdir(os, obj, tx, RW_WRITER,
603             B_FALSE, B_FALSE, &zap));
604         VERIFY0(mzap_upgrade(&zap, tx, flags));
604         VERIFY3U(0, ==, mzap_upgrade(&zap, tx, flags));
605         zap_unlockdir(zap);
606     }
607 }
_____unchanged_portion_omitted_____

```

```
*****
14191 Thu Jul 26 14:19:26 2012
new/usr/src/uts/common/fs/zfs/zfeature.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

342 /*
343  * Enable any required dependencies, then enable the requested feature.
344  */
345 void
346 spa_feature_enable(spa_t *spa, zfeature_info_t *feature, dmu_tx_t *tx)
347 {
348     ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);
349     VERIFY0(feature_do_action(spa->spa_meta_objset,
350     VERIFY3U(0, ==, feature_do_action(spa->spa_meta_objset,
351     spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
352     spa->spa_feat_desc_obj, feature, FEATURE_ACTION_ENABLE, tx));
352 }

354 /*
355  * If the specified feature has not yet been enabled, this function returns
356  * ENOTSUP; otherwise, this function increments the feature's refcount (or
357  * returns EOVERFLOW if the refcount cannot be incremented). This function must
358  * be called from syncing context.
359  */
360 void
361 spa_feature_incr(spa_t *spa, zfeature_info_t *feature, dmu_tx_t *tx)
362 {
363     ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);
364     VERIFY0(feature_do_action(spa->spa_meta_objset,
365     VERIFY3U(0, ==, feature_do_action(spa->spa_meta_objset,
366     spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
367     spa->spa_feat_desc_obj, feature, FEATURE_ACTION_INCR, tx));
367 }

369 /*
370  * If the specified feature has not yet been enabled, this function returns
371  * ENOTSUP; otherwise, this function decrements the feature's refcount (or
372  * returns EOVERFLOW if the refcount is already 0). This function must
373  * be called from syncing context.
374  */
375 void
376 spa_feature_decr(spa_t *spa, zfeature_info_t *feature, dmu_tx_t *tx)
377 {
378     ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);
379     VERIFY0(feature_do_action(spa->spa_meta_objset,
380     VERIFY3U(0, ==, feature_do_action(spa->spa_meta_objset,
381     spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
382     spa->spa_feat_desc_obj, feature, FEATURE_ACTION_DECR, tx));
382 }
_____unchanged_portion_omitted_____
```

new/usr/src/uts/common/fs/zfs/zfs\_debug.c

1

```
*****
2598 Thu Jul 26 14:19:26 2012
new/usr/src/uts/common/fs/zfs/zfs_debug.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
```

```
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) 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright (c) 2012 by Delphix. All rights reserved.
24 */
```

```
27 #include <sys/zfs_context.h>
```

```
29 list_t zfs_dbgmsgs;
30 int zfs_dbgmsg_size;
31 kmutex_t zfs_dbgmsgs_lock;
32 int zfs_dbgmsg_maxsize = 1<<20; /* 1MB */
```

```
34 void
35 zfs_dbgmsg_init(void)
36 {
37     list_create(&zfs_dbgmsgs, sizeof(zfs_dbgmsg_t),
38               offsetof(zfs_dbgmsg_t, zdm_node));
39     mutex_init(&zfs_dbgmsgs_lock, NULL, MUTEX_DEFAULT, NULL);
40 }
42 void
43 zfs_dbgmsg_fini(void)
44 {
45     zfs_dbgmsg_t *zdm;
47     while ((zdm = list_remove_head(&zfs_dbgmsgs)) != NULL) {
48         int size = sizeof(zfs_dbgmsg_t) + strlen(zdm->zdm_msg);
49         kmem_free(zdm, size);
50         zfs_dbgmsg_size -= size;
51     }
52     mutex_destroy(&zfs_dbgmsgs_lock);
53     ASSERT0(zfs_dbgmsg_size);
54     ASSERT3U(zfs_dbgmsg_size, ==, 0);
55 }
```

unchanged\_portion\_omitted

```

*****
29716 Thu Jul 26 14:19:27 2012
new/usr/src/uts/common/fs/zfs/zfs_dir.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 #include <sys/types.h>
27 #include <sys/param.h>
28 #include <sys/time.h>
29 #include <sys/sysm.h>
30 #include <sys/sysmacros.h>
31 #include <sys/resource.h>
32 #include <sys/vfs.h>
33 #include <sys/vnode.h>
34 #include <sys/file.h>
35 #include <sys/mode.h>
36 #include <sys/kmem.h>
37 #include <sys/uio.h>
38 #include <sys/pathname.h>
39 #include <sys/cmn_err.h>
40 #include <sys/errno.h>
41 #include <sys/stat.h>
42 #include <sys/unistd.h>
43 #include <sys/sunddi.h>
44 #include <sys/random.h>
45 #include <sys/policy.h>
46 #include <sys/zfs_dir.h>
47 #include <sys/zfs_acl.h>
48 #include <sys/fs/zfs.h>
49 #include "fs/fs_subr.h"
50 #include <sys/zap.h>
51 #include <sys/dmu.h>
52 #include <sys/atomic.h>
53 #include <sys/zfs_ctldir.h>
54 #include <sys/zfs_fuid.h>
55 #include <sys/sa.h>
56 #include <sys/zfs_sa.h>
57 #include <sys/dnld.h>
58 #include <sys/extdirent.h>

60 /*
61 * zfs_match_find() is used by zfs_dirent_lock() to perform zap lookups

```

```

62 * of names after deciding which is the appropriate lookup interface.
63 */
64 static int
65 zfs_match_find(zfsvfs_t *zfsvfs, znode_t *dzp, char *name, boolean_t exact,
66              boolean_t update, int *deflags, pathname_t *rpn, uint64_t *zoid)
67 {
68     int error;

70     if (zfsvfs->z_norm) {
71         matchtype_t mt = MT_FIRST;
72         boolean_t conflict = B_FALSE;
73         size_t bufisz = 0;
74         char *buf = NULL;

76         if (rpn) {
77             buf = rpn->pn_buf;
78             bufisz = rpn->pn_bufsize;
79         }
80         if (exact)
81             mt = MT_EXACT;
82         /*
83          * In the non-mixed case we only expect there would ever
84          * be one match, but we need to use the normalizing lookup.
85          */
86         error = zap_lookup_norm(zfsvfs->z_os, dzp->z_id, name, 8, 1,
87                               zoid, mt, buf, bufisz, &conflict);
88         if (!error && deflags)
89             *deflags = conflict ? ED_CASE_CONFLICT : 0;
90     } else {
91         error = zap_lookup(zfsvfs->z_os, dzp->z_id, name, 8, 1, zoid);
92     }
93     *zoid = ZFS_DIRENT_OBJ(*zoid);

95     if (error == ENOENT && update)
96         dnld_update(ZTOV(dzp), name, DNLC_NO_VNODE);

98     return (error);
99 }
_____ unchanged_portion_omitted _____

436 /*
437 * unlinked Set (formerly known as the "delete queue") Error Handling
438 *
439 * When dealing with the unlinked set, we dm_u_tx_hold_zap(), but we
440 * don't specify the name of the entry that we will be manipulating. We
441 * also fib and say that we won't be adding any new entries to the
442 * unlinked set, even though we might (this is to lower the minimum file
443 * size that can be deleted in a full filesystem). So on the small
444 * chance that the nlink list is using a fat zap (ie. has more than
445 * 2000 entries), we *may* not pre-read a block that's needed.
446 * Therefore it is remotely possible for some of the assertions
447 * regarding the unlinked set below to fail due to i/o error. On a
448 * nondebug system, this will result in the space being leaked.
449 */
450 void
451 zfs_unlinked_add(znode_t *zp, dm_u_tx_t *tx)
452 {
453     zfsvfs_t *zfsvfs = zp->z_zfsvfs;

455     ASSERT(zp->z_unlinked);
456     ASSERT(zp->z_links == 0);

458     VERIFY0(zap_add_int(zfsvfs->z_os, zfsvfs->z_unlinkedobj,
459                       zp->z_id, tx));
457     VERIFY3U(0, ==,
458             zap_add_int(zfsvfs->z_os, zfsvfs->z_unlinkedobj, zp->z_id, tx));

```

```

460 }
    unchanged_portion_omitted

581 void
582 zfs_rmnode(znode_t *zp)
583 {
584     zfsvfs_t      *zfsvfs = zp->z_zfsvfs;
585     objset_t      *os = zfsvfs->z_os;
586     znode_t       *xzp = NULL;
587     dmu_tx_t      *tx;
588     uint64_t      acl_obj;
589     uint64_t      xattr_obj;
590     int           error;

592     ASSERT(zp->z_links == 0);
593     ASSERT(ZTOV(zp)->v_count == 0);

595     /*
596      * If this is an attribute directory, purge its contents.
597      */
598     if (ZTOV(zp)->v_type == VDIR && (zp->z_pflags & ZFS_XATTR)) {
599         if (zfs_purgedir(zp) != 0) {
600             /*
601              * Not enough space to delete some xattrs.
602              * Leave it in the unlinked set.
603              */
604             zfs_znode_dmu_fini(zp);
605             zfs_znode_free(zp);
606             return;
607         }
608     }

610     /*
611      * Free up all the data in the file.
612      */
613     error = dmu_free_long_range(os, zp->z_id, 0, DMU_OBJECT_END);
614     if (error) {
615         /*
616          * Not enough space. Leave the file in the unlinked set.
617          */
618         zfs_znode_dmu_fini(zp);
619         zfs_znode_free(zp);
620         return;
621     }

623     /*
624      * If the file has extended attributes, we're going to unlink
625      * the xattr dir.
626      */
627     error = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
628         &xattr_obj, sizeof(xattr_obj));
629     if (error == 0 && xattr_obj) {
630         error = zfs_zget(zfsvfs, xattr_obj, &xzp);
631         ASSERT(error == 0);
632     }

634     acl_obj = zfs_external_acl(zp);

636     /*
637      * Set up the final transaction.
638      */
639     tx = dmu_tx_create(os);
640     dmu_tx_hold_free(tx, zp->z_id, 0, DMU_OBJECT_END);
641     dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
642     if (xzp) {
643         dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, TRUE, NULL);

```

```

644         dmu_tx_hold_sa(tx, xzp->z_sa_hdl, B_FALSE);
645     }
646     if (acl_obj)
647         dmu_tx_hold_free(tx, acl_obj, 0, DMU_OBJECT_END);

649     zfs_sa_upgrade_txholds(tx, zp);
650     error = dmu_tx_assign(tx, TXG_WAIT);
651     if (error) {
652         /*
653          * Not enough space to delete the file. Leave it in the
654          * unlinked set, leaking it until the fs is remounted (at
655          * which point we'll call zfs_unlinked_drain() to process it).
656          */
657         dmu_tx_abort(tx);
658         zfs_znode_dmu_fini(zp);
659         zfs_znode_free(zp);
660         goto out;
661     }

663     if (xzp) {
664         ASSERT(error == 0);
665         mutex_enter(&xzp->z_lock);
666         xzp->z_unlinked = B_TRUE; /* mark xzp for deletion */
667         xzp->z_links = 0; /* no more links to it */
668         VERIFY(0 == sa_update(xzp->z_sa_hdl, SA_ZPL_LINKS(zfsvfs),
669             &xzp->z_links, sizeof(xzp->z_links), tx));
670         mutex_exit(&xzp->z_lock);
671         zfs_unlinked_add(xzp, tx);
672     }

674     /* Remove this znode from the unlinked set */
675     VERIFY0(zap_remove_int(zfsvfs->z_os, zfsvfs->z_unlinkedobj,
676         xzp->z_id, tx));
677     VERIFY3U(0, ==,
678         zap_remove_int(zfsvfs->z_os, zfsvfs->z_unlinkedobj, zp->z_id, tx));

678     zfs_znode_delete(zp, tx);

680     dmu_tx_commit(tx);
681 out:
682     if (xzp)
683         VN_RELE(ZTOV(xzp));
684 }
    unchanged_portion_omitted

```

new/usr/src/uts/common/fs/zfs/zfs\_ioctl.c

1

```
*****
142364 Thu Jul 26 14:19:28 2012
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

1938 static int
1939 zfs_ioc_objset_stats_impl(zfs_cmd_t *zc, objset_t *os)
1940 {
1941     int error = 0;
1942     nvlist_t *nv;

1944     dmu_objset_fast_stat(os, &zc->zc_objset_stats);

1946     if (zc->zc_nvlist_dst != 0 &&
1947         (error = dsl_prop_get_all(os, &nv)) == 0) {
1948         dmu_objset_stats(os, nv);
1949         /*
1950          * NB: zvol_get_stats() will read the objset contents,
1951          * which we aren't supposed to do with a
1952          * DS_MODE_USER hold, because it could be
1953          * inconsistent. So this is a bit of a workaround...
1954          * XXX reading with out owning
1955          */
1956         if (!zc->zc_objset_stats.dds_inconsistent &&
1957             dmu_objset_type(os) == DMU_OST_ZVOL) {
1958             error = zvol_get_stats(os, nv);
1959             if (error == EIO)
1960                 return (error);
1961             VERIFY0(error);
1961             VERIFY3S(error, ==, 0);
1962         }
1963         error = put_nvlist(zc, nv);
1964         nvlist_free(nv);
1965     }

1967     return (error);
1968 }
_____unchanged_portion_omitted_____
```



```

*****
17055 Thu Jul 26 14:19:29 2012
new/usr/src/uts/common/fs/zfs/zfs_rlock.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 /*
27 * Copyright (c) 2012 by Delphix. All rights reserved.
28 */

32 /*
33 * This file contains the code to implement file range locking in
34 * ZFS, although there isn't much specific to ZFS (all that comes to mind
35 * support for growing the blocksize).
36 *
37 * Interface
38 * -----
39 * Defined in zfs_rlock.h but essentially:
40 *   rl = zfs_range_lock(zp, off, len, lock_type);
41 *   zfs_range_unlock(rl);
42 *   zfs_range_reduce(rl, off, len);
43 *
44 * AVL tree
45 * -----
46 * An AVL tree is used to maintain the state of the existing ranges
47 * that are locked for exclusive (writer) or shared (reader) use.
48 * The starting range offset is used for searching and sorting the tree.
49 *
50 * Common case
51 * -----
52 * The (hopefully) usual case is of no overlaps or contention for
53 * locks. On entry to zfs_lock_range() a rl_t is allocated; the tree
54 * searched that finds no overlap, and *this* rl_t is placed in the tree.
55 *
56 * Overlaps/Reference counting/Proxy locks
57 * -----
58 * The avl code only allows one node at a particular offset. Also it's very
59 * inefficient to search through all previous entries looking for overlaps
60 * (because the very 1st in the ordered list might be at offset 0 but
61 * cover the whole file).

```

```

62 * So this implementation uses reference counts and proxy range locks.
63 * Firstly, only reader locks use reference counts and proxy locks,
64 * because writer locks are exclusive.
65 * When a reader lock overlaps with another then a proxy lock is created
66 * for that range and replaces the original lock. If the overlap
67 * is exact then the reference count of the proxy is simply incremented.
68 * Otherwise, the proxy lock is split into smaller lock ranges and
69 * new proxy locks created for non overlapping ranges.
70 * The reference counts are adjusted accordingly.
71 * Meanwhile, the original lock is kept around (this is the callers handle)
72 * and its offset and length are used when releasing the lock.
73 *
74 * Thread coordination
75 * -----
76 * In order to make wakeups efficient and to ensure multiple continuous
77 * readers on a range don't starve a writer for the same range lock,
78 * two condition variables are allocated in each rl_t.
79 * If a writer (or reader) can't get a range it initialises the writer
80 * (or reader) cv; sets a flag saying there's a writer (or reader) waiting;
81 * and waits on that cv. When a thread unlocks that range it wakes up all
82 * writers then all readers before destroying the lock.
83 *
84 * Append mode writes
85 * -----
86 * Append mode writes need to lock a range at the end of a file.
87 * The offset of the end of the file is determined under the
88 * range locking mutex, and the lock type converted from RL_APPEND to
89 * RL_WRITER and the range locked.
90 *
91 * Grow block handling
92 * -----
93 * ZFS supports multiple block sizes currently upto 128K. The smallest
94 * block size is used for the file which is grown as needed. During this
95 * growth all other writers and readers must be excluded.
96 * So if the block size needs to be grown then the whole file is
97 * exclusively locked, then later the caller will reduce the lock
98 * range to just the range to be written using zfs_reduce_range.
99 */

101 #include <sys/zfs_rlock.h>

103 /*
104 * Check if a write lock can be grabbed, or wait and recheck until available.
105 */
106 static void
107 zfs_range_lock_writer(znode_t *zp, rl_t *new)
108 {
109     avl_tree_t *tree = &zp->z_range_avl;
110     rl_t *rl;
111     avl_index_t where;
112     uint64_t end_size;
113     uint64_t off = new->r_off;
114     uint64_t len = new->r_len;

116     for (;;) {
117         /*
118          * Range locking is also used by zvol and uses a
119          * dummed up znode. However, for zvol, we don't need to
120          * append or grow blocksize, and besides we don't have
121          * a "sa" data or z_zfsvfs - so skip that processing.
122          *
123          * Yes, this is ugly, and would be solved by not handling
124          * grow or append in range lock code. If that was done then
125          * we could make the range locking code generically available
126          * to other non-zfs consumers.
127          */

```

```

128     if (zp->z_vnode) { /* caller is ZPL */
129         /*
130          * If in append mode pick up the current end of file.
131          * This is done under z_range_lock to avoid races.
132          */
133         if (new->r_type == RL_APPEND)
134             new->r_off = zp->z_size;
135
136         /*
137          * If we need to grow the block size then grab the whole
138          * file range. This is also done under z_range_lock to
139          * avoid races.
140          */
141         end_size = MAX(zp->z_size, new->r_off + len);
142         if (end_size > zp->z_blkisz && (!ISP2(zp->z_blkisz) ||
143             zp->z_blkisz < zp->z_zfsvfs->z_max_blkisz)) {
144             new->r_off = 0;
145             new->r_len = UINT64_MAX;
146         }
147     }
148
149     /*
150     * First check for the usual case of no locks
151     */
152     if (avl_numnodes(tree) == 0) {
153         new->r_type = RL_WRITER; /* convert to writer */
154         avl_add(tree, new);
155         return;
156     }
157
158     /*
159     * Look for any locks in the range.
160     */
161     rl = avl_find(tree, new, &where);
162     if (rl)
163         goto wait; /* already locked at same offset */
164
165     rl = (rl_t *)avl_nearest(tree, where, AVL_AFTER);
166     if (rl && (rl->r_off < new->r_off + new->r_len))
167         goto wait;
168
169     rl = (rl_t *)avl_nearest(tree, where, AVL_BEFORE);
170     if (rl && rl->r_off + rl->r_len > new->r_off)
171         goto wait;
172
173     new->r_type = RL_WRITER; /* convert possible RL_APPEND */
174     avl_insert(tree, new, where);
175     return;
176 wait:
177     if (!rl->r_write_wanted) {
178         cv_init(&rl->r_wr_cv, NULL, CV_DEFAULT, NULL);
179         rl->r_write_wanted = B_TRUE;
180     }
181     cv_wait(&rl->r_wr_cv, &zp->z_range_lock);
182
183     /* reset to original */
184     new->r_off = off;
185     new->r_len = len;
186 }
187 }

```

unchanged portion omitted

```

462 /*
463  * Unlock a reader lock
464  */
465 static void

```

```

466 zfs_range_unlock_reader(znode_t *zp, rl_t *remove)
467 {
468     avl_tree_t *tree = &zp->z_range_avl;
469     rl_t *rl, *next;
470     uint64_t len;
471
472     /*
473     * The common case is when the remove entry is in the tree
474     * (cnt == 1) meaning there's been no other reader locks overlapping
475     * with this one. Otherwise the remove entry will have been
476     * removed from the tree and replaced by proxies (one or
477     * more ranges mapping to the entire range).
478     */
479     if (remove->r_cnt == 1) {
480         avl_remove(tree, remove);
481         if (remove->r_write_wanted) {
482             cv_broadcast(&remove->r_wr_cv);
483             cv_destroy(&remove->r_wr_cv);
484         }
485         if (remove->r_read_wanted) {
486             cv_broadcast(&remove->r_rd_cv);
487             cv_destroy(&remove->r_rd_cv);
488         }
489     } else {
490         ASSERT0(remove->r_cnt);
491         ASSERT0(remove->r_write_wanted);
492         ASSERT0(remove->r_read_wanted);
493         ASSERT3U(remove->r_cnt, ==, 0);
494         ASSERT3U(remove->r_write_wanted, ==, 0);
495         ASSERT3U(remove->r_read_wanted, ==, 0);
496         /*
497          * Find start proxy representing this reader lock,
498          * then decrement ref count on all proxies
499          * that make up this range, freeing them as needed.
500          */
501         rl = avl_find(tree, remove, NULL);
502         ASSERT(rl);
503         ASSERT(rl->r_cnt);
504         ASSERT(rl->r_type == RL_READER);
505         for (len = remove->r_len; len != 0; rl = next) {
506             len -= rl->r_len;
507             if (len) {
508                 next = AVL_NEXT(tree, rl);
509                 ASSERT(next);
510                 ASSERT(rl->r_off + rl->r_len == next->r_off);
511                 ASSERT(next->r_cnt);
512                 ASSERT(next->r_type == RL_READER);
513             }
514             rl->r_cnt--;
515             if (rl->r_cnt == 0) {
516                 avl_remove(tree, rl);
517                 if (rl->r_write_wanted) {
518                     cv_broadcast(&rl->r_wr_cv);
519                     cv_destroy(&rl->r_wr_cv);
520                 }
521                 if (rl->r_read_wanted) {
522                     cv_broadcast(&rl->r_rd_cv);
523                     cv_destroy(&rl->r_rd_cv);
524                 }
525                 kmem_free(rl, sizeof(rl_t));
526             }
527         }
528         kmem_free(remove, sizeof(rl_t));
529     }

```

unchanged portion omitted

new/usr/src/uts/common/fs/zfs/zfs\_vfsops.c

1

```
*****
58641 Thu Jul 26 14:19:30 2012
new/usr/src/uts/common/fs/zfs/zfs_vfsops.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

2198 int
2199 zfs_set_version(zfsvfs_t *zfsvfs, uint64_t newvers)
2200 {
2201     int error;
2202     objset_t *os = zfsvfs->z_os;
2203     dmu_tx_t *tx;

2205     if (newvers < ZPL_VERSION_INITIAL || newvers > ZPL_VERSION)
2206         return (EINVAL);

2208     if (newvers < zfsvfs->z_version)
2209         return (EINVAL);

2211     if (zfs_spa_version_map(newvers) >
2212         spa_version(dmu_objset_spa(zfsvfs->z_os)))
2213         return (ENOTSUP);

2215     tx = dmu_tx_create(os);
2216     dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, B_FALSE, ZPL_VERSION_STR);
2217     if (newvers >= ZPL_VERSION_SA && !zfsvfs->z_use_sa) {
2218         dmu_tx_hold_zap(tx, MASTER_NODE_OBJ, B_TRUE,
2219             ZFS_SA_ATTRS);
2220         dmu_tx_hold_zap(tx, DMU_NEW_OBJECT, FALSE, NULL);
2221     }
2222     error = dmu_tx_assign(tx, TXG_WAIT);
2223     if (error) {
2224         dmu_tx_abort(tx);
2225         return (error);
2226     }

2228     error = zap_update(os, MASTER_NODE_OBJ, ZPL_VERSION_STR,
2229         8, 1, &newvers, tx);

2231     if (error) {
2232         dmu_tx_commit(tx);
2233         return (error);
2234     }

2236     if (newvers >= ZPL_VERSION_SA && !zfsvfs->z_use_sa) {
2237         uint64_t sa_obj;

2239         ASSERT3U(spa_version(dmu_objset_spa(zfsvfs->z_os)), >=,
2240             SPA_VERSION_SA);
2241         sa_obj = zap_create(os, DMU_OT_SA_MASTER_NODE,
2242             DMU_OT_NONE, 0, tx);

2244         error = zap_add(os, MASTER_NODE_OBJ,
2245             ZFS_SA_ATTRS, 8, 1, &sa_obj, tx);
2246         ASSERT0(error);
2247         ASSERT3U(error, ==, 0);

2248         VERIFY(0 == sa_set_sa_object(os, sa_obj));
2249         sa_register_update_callback(os, zfs_sa_upgrade);
2250     }

2252     spa_history_log_internal_ds(dmu_objset_ds(os), "upgrade", tx,
2253         "from %llu to %llu", zfsvfs->z_version, newvers);

2255     dmu_tx_commit(tx);
```

new/usr/src/uts/common/fs/zfs/zfs\_vfsops.c

2

```
2257         zfsvfs->z_version = newvers;
2259         zfs_set_fuid_feature(zfsvfs);

2261         return (0);
2262     }
_____unchanged_portion_omitted_____
```

```

*****
129321 Thu Jul 26 14:19:31 2012
new/usr/src/uts/common/fs/zfs/zfs_vnops.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 */

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

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

```

```

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

86 /*
87  * Programming rules.
88  *
89  * Each vnode op performs some logical unit of work. To do this, the ZPL must
90  * properly lock its in-core state, create a DMU transaction, do the work,
91  * record this work in the intent log (ZIL), commit the DMU transaction,
92  * and wait for the intent log to commit if it is a synchronous operation.
93  * Moreover, the vnode ops must work in both normal and log replay context.
94  * The ordering of events is important to avoid deadlocks and references
95  * to freed memory. The example below illustrates the following Big Rules:
96  *
97  * (1) A check must be made in each zfs thread for a mounted file system.
98  * This is done avoiding races using ZFS_ENTER(zfsvfs).
99  * A ZFS_EXIT(zfsvfs) is needed before all returns. Any znodes
100 * must be checked with ZFS_VERIFY_ZP(zp). Both of these macros
101 * can return EIO from the calling function.
102 *
103 * (2) VN_RELE() should always be the last thing except for zil_commit()
104 * (if necessary) and ZFS_EXIT(). This is for 3 reasons:
105 * First, if it's the last reference, the vnode/znnode
106 * can be freed, so the zp may point to freed memory. Second, the last
107 * reference will call zfs_zinactive(), which may induce a lot of work --
108 * pushing cached pages (which acquires range locks) and syncing out
109 * cached atime changes. Third, zfs_zinactive() may require a new tx,
110 * which could deadlock the system if you were already holding one.
111 * If you must call VN_RELE() within a tx then use VN_RELE_ASYNC().
112 *
113 * (3) All range locks must be grabbed before calling dmu_tx_assign(),
114 * as they can span dmu_tx_assign() calls.
115 *
116 * (4) Always pass TXG_NOWAIT as the second argument to dmu_tx_assign().
117 * This is critical because we don't want to block while holding locks.
118 * Note, in particular, that if a lock is sometimes acquired before
119 * the tx assigns, and sometimes after (e.g. z_lock), then failing to
120 * use a non-blocking assign can deadlock the system. The scenario:
121 *
122 * Thread A has grabbed a lock before calling dmu_tx_assign().
123 * Thread B is in an already-assigned tx, and blocks for this lock.
124 * Thread A calls dmu_tx_assign(TXG_WAIT) and blocks in txg_wait_open()
125 * forever, because the previous txg can't quiesce until B's tx commits.
126 *
127 * If dmu_tx_assign() returns ERESTART and zfsvfs->z_assign is TXG_NOWAIT,

```

```

128 *      then drop all locks, call dm_u_tx_wait(), and try again.
129 *
130 * (5) If the operation succeeded, generate the intent log entry for it
131 * before dropping locks. This ensures that the ordering of events
132 * in the intent log matches the order in which they actually occurred.
133 * During ZIL replay the zfs_log_* functions will update the sequence
134 * number to indicate the zil transaction has replayed.
135 *
136 * (6) At the end of each vnode op, the DMU tx must always commit,
137 * regardless of whether there were any errors.
138 *
139 * (7) After dropping all locks, invoke zil_commit(zilog, foid)
140 * to ensure that synchronous semantics are provided when necessary.
141 *
142 * In general, this is how things should be ordered in each vnode op:
143 *
144 *      ZFS_ENTER(zfsvfs);          // exit if unmounted
145 * top:
146 *      zfs_dirent_lock(&dl, ...)   // lock directory entry (may VN_HOLD())
147 *      rw_enter(...);             // grab any other locks you need
148 *      tx = dm_u_tx_create(...);   // get DMU tx
149 *      dm_u_tx_hold_*();           // hold each object you might modify
150 *      error = dm_u_tx_assign(tx, TXG_NOWAIT); // try to assign
151 *      if (error) {
152 *          rw_exit(...);          // drop locks
153 *          zfs_dirent_unlock(dl);  // unlock directory entry
154 *          VN_RELE(...);         // release held vnodes
155 *          if (error == ERESTART) {
156 *              dm_u_tx_wait(tx);
157 *              dm_u_tx_abort(tx);
158 *              goto top;
159 *          }
160 *          dm_u_tx_abort(tx);      // abort DMU tx
161 *          ZFS_EXIT(zfsvfs);       // finished in zfs
162 *          return (error);        // really out of space
163 *      }
164 *      error = do_real_work();     // do whatever this VOP does
165 *      if (error == 0)
166 *          zfs_log*(...);         // on success, make ZIL entry
167 *      dm_u_tx_commit(tx);        // commit DMU tx -- error or not
168 *      rw_exit(...);            // drop locks
169 *      zfs_dirent_unlock(dl);     // unlock directory entry
170 *      VN_RELE(...);            // release held vnodes
171 *      zil_commit(zilog, foid);   // synchronous when necessary
172 *      ZFS_EXIT(zfsvfs);         // finished in zfs
173 *      return (error);           // done, report error
174 */
176 /* ARGSUSED */
177 static int
178 zfs_open(vnode_t **vpp, int flag, cred_t *cr, caller_context_t *ct)
179 {
180     vnode_t *zp = VTOZ(*vpp);
181     zfsvfs_t *zfsvfs = zp->z_zfsvfs;
183     ZFS_ENTER(zfsvfs);
184     ZFS_VERIFY_ZP(zp);
186     if ((flag & FWRITE) && (zp->z_pflags & ZFS_APPENDONLY) &&
187         ((flag & FAPPEND) == 0)) {
188         ZFS_EXIT(zfsvfs);
189         return (EPERM);
190     }
192     if (!zfs_has_ctldir(zp) && zp->z_zfsvfs->z_vscan &&
193         ZTOV(zp)->v_type == VREG &&

```

```

194         !(zp->z_pflags & ZFS_AV_QUARANTINED) && zp->z_size > 0) {
195             if (fs_vscan(*vpp, cr, 0) != 0) {
196                 ZFS_EXIT(zfsvfs);
197                 return (EACCES);
198             }
199         }
201     /* Keep a count of the synchronous opens in the znode */
202     if (flag & (FSYNC | FDSYNC))
203         atomic_inc_32(&zp->z_sync_cnt);
205     ZFS_EXIT(zfsvfs);
206     return (0);
207 }
208 }
209 }
210 }
211 }
212 }
213 }
214 }
215 }
216 }
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }
225 }
226 }
227 }
228 }
229 }
230 }
231 }
232 }
233 }
234 }
235 }
236 }
237 }
238 }
239 }
240 }
241 }
242 }
243 }
244 }
245 }
246 }
247 }
248 }
249 }
250 }
251 }
252 }
253 }
254 }
255 }
256 }
257 }
258 }
259 }
260 }
261 }
262 }
263 }
264 }
265 }
266 }
267 }
268 }
269 }
270 }
271 }
272 }
273 }
274 }
275 }
276 }
277 }
278 }
279 }
280 }
281 }
282 }
283 }
284 }
285 }
286 }
287 }
288 }
289 }
290 }
291 }
292 }
293 }
294 }
295 }
296 }
297 }
298 }
299 }
300 }
301 }
302 }
303 }
304 }
305 }
306 }
307 }
308 }
309 }
310 }
311 }
312 }
313 }
314 }
315 }
316 }
317 }
318 }
319 }
320 }
321 }
322 }
323 }
324 }
325 }
326 }
327 }
328 }
329 }
330 }
331 }
332 }
333 }
334 }
335 }
336 }
337 }
338 }
339 }
340 }
341 }
342 }
343 }
344 }
345 }
346 }
347 }
348 }
349 }
350 }
351 }
352 }
353 }
354 }
355 }
356 }
357 }
358 }
359 }
360 }
361 }
362 }
363 }
364 }
365 }
366 }
367 }
368 }
369 }
370 }
371 }
372 }
373 }
374 }
375 }
376 }
377 }
378 }
379 }
380 }
381 }
382 }
383 }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }
393 }
394 }
395 }
396 }
397 }
398 }
399 }
400 }
401 }
402 }
403 }
404 }
405 }
406 }
407 }
408 }
409 }
410 }
411 }
412 }
413 }
414 }
415 }
416 }
417 }
418 }
419 }
420 }
421 }
422 }
423 }
424 }
425 }
426 }
427 }
428 }
429 }
430 }
431 }
432 }
433 }
434 }
435 }
436 }
437 }
438 }
439 }
440 }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }
449 }
450 }
451 }
452 }
453 }
454 }
455 }
456 }
457 }
458 }
459 }
460 }
461 }
462 }
463 }
464 }
465 }
466 }
467 }
468 }
469 }
470 }
471 }
472 }
473 }
474 }
475 }
476 }
477 }
478 }
479 }
480 }
481 }
482 }
483 }
484 }
485 }
486 }
487 }
488 }
489 }
490 }
491 }
492 }
493 }
494 }
495 }
496 }
497 }
498 }
499 }
500 }
501 }
502 }
503 }
504 }
505 }
506 }
507 }
508 }
509 }
510 }
511 }
512 }
513 }
514 }
515 }
516 }
517 }
518 }
519 }
520 }
521 }
522 }
523 }
524 }
525 }
526 }
527 }
528 }
529 }
530 }
531 }
532 }
533 }
534 }
535 }
536 }
537 }
538 }
539 }
540 }
541 }
542 }
543 }
544 }
545 }
546 }
547 }
548 }
549 }
550 }
551 }
552 }
553 }
554 }
555 }
556 }
557 }
558 }
559 }
560 }
561 }
562 }
563 }
564 }
565 }
566 }
567 }
568 }
569 }
570 }
571 }
572 }
573 }
574 }
575 }
576 }
577 }
578 }
579 }
580 }
581 }
582 }
583 }
584 }
585 }
586 }
587 }
588 }
589 }
590 }
591 }
592 }
593 }
594 }
595 }
596 }
597 }
598 }
599 }
600 }
601 }
602 }
603 }
604 }
605 }
606 }
607 }
608 }
609 }
610 }
611 }
612 }
613 }
614 }
615 }
616 }
617 }
618 }
619 }
620 }
621 }
622 }
623 }
624 }
625 }
626 }
627 }
628 }
629 }
630 }
631 }
632 }
633 }
634 }
635 }
636 }
637 }
638 }
639 }
640 }
641 }
642 }
643 }
644 }
645 }
646 }
647 }
648 }
649 }
650 }
651 }
652 }
653 }
654 }
655 }
656 }
657 }
658 }
659 }
660 }
661 }
662 }
663 }
664 }
665 }
666 }
667 }
668 }
669 }
670 }
671 }
672 }
673 }
674 }
675 }
676 }
677 }
678 }
679 }
680 }
681 }
682 }
683 }
684 }
685 }
686 }
687 }
688 }
689 }
690 }
691 }
692 }
693 }
694 }
695 }
696 }
697 }
698 }
699 }
700 }
701 }
702 }
703 }
704 }
705 }
706 }
707 }
708 }
709 }
710 }
711 }
712 }
713 }
714 }
715 }
716 }
717 }
718 }
719 }
720 }
721 }
722 }
723 }
724 }
725 }
726 }
727 }
728 }
729 }
730 }
731 }
732 }
733 }
734 }
735 }
736 }
737 }
738 }
739 }
740 }
741 }
742 }
743 }
744 }
745 }
746 }
747 }
748 }
749 }
750 }
751 }
752 }
753 }
754 }
755 }
756 }
757 }
758 }
759 }
760 }
761 }
762 }
763 }
764 }
765 }
766 }
767 }
768 }
769 }
770 }
771 }
772 }
773 }
774 }
775 }
776 }
777 }
778 }
779 }
780 }
781 }
782 }
783 }
784 }
785 }
786 }
787 }
788 }
789 }
790 }
791 }
792 }
793 }
794 }
795 }
796 }
797 }
798 }
799 }
800 }
801 }
802 }
803 }
804 }
805 }
806 }
807 }
808 }
809 }
810 }
811 }
812 }
813 }
814 }
815 }
816 }
817 }
818 }
819 }
820 }
821 }
822 }
823 }
824 }
825 }
826 }
827 }
828 }
829 }
830 }
831 }
832 }
833 }
834 }
835 }
836 }
837 }
838 }
839 }
840 }
841 }
842 }
843 }
844 }
845 }
846 }
847 }
848 }
849 }
850 }
851 }
852 }
853 }
854 }
855 }
856 }
857 }
858 }
859 }
860 }
861 }
862 }
863 }
864 }
865 }
866 }
867 }
868 }
869 }
870 }
871 }
872 }
873 }
874 }
875 }
876 }
877 }
878 }
879 }
880 }
881 }
882 }
883 }
884 }
885 }
886 }
887 }
888 }
889 }
890 }
891 }
892 }
893 }
894 }
895 }
896 }
897 }
898 }
899 }
900 }
901 }
902 }
903 }
904 }
905 }
906 }
907 }
908 }
909 }
910 }
911 }
912 }
913 }
914 }
915 }
916 }
917 }
918 }
919 }
920 }
921 }
922 }
923 }
924 }
925 }
926 }
927 }
928 }
929 }
930 }
931 }
932 }
933 }
934 }
935 }
936 }
937 }
938 }
939 }
940 }
941 }
942 }
943 }
944 }
945 }
946 }
947 }
948 }
949 }
950 }
951 }
952 }
953 }
954 }
955 }
956 }
957 }
958 }
959 }
960 }
961 }
962 }
963 }
964 }
965 }
966 }
967 }
968 }
969 }
970 }
971 }
972 }
973 }
974 }
975 }
976 }
977 }
978 }
979 }
980 }
981 }
982 }
983 }
984 }
985 }
986 }
987 }
988 }
989 }
990 }
991 }
992 }
993 }
994 }
995 }
996 }
997 }
998 }
999 }
1000 }

```

```

1582     }
1584 top:
1585     xattr_obj = 0;
1586     xzp = NULL;
1587     /*
1588      * Attempt to lock directory; fail if entry doesn't exist.
1589      */
1590     if (error = zfs_dirent_lock(&dl, dzp, name, &zp, zflg,
1591         NULL, realnmp)) {
1592         if (realnmp)
1593             pn_free(realnmp);
1594         ZFS_EXIT(zfsvfs);
1595         return (error);
1596     }
1598     vp = ZTOV(zp);
1600     if (error = zfs_zaccess_delete(dzp, zp, cr)) {
1601         goto out;
1602     }
1604     /*
1605      * Need to use rmdir for removing directories.
1606      */
1607     if (vp->v_type == VDIR) {
1608         error = EPERM;
1609         goto out;
1610     }
1612     vnevent_remove(vp, dvp, name, ct);
1614     if (realnmp)
1615         dnlc_remove(dvp, realnmp->pn_buf);
1616     else
1617         dnlc_remove(dvp, name);
1619     mutex_enter(&vp->v_lock);
1620     may_delete_now = vp->v_count == 1 && !vn_has_cached_data(vp);
1621     mutex_exit(&vp->v_lock);
1623     /*
1624      * We may delete the znode now, or we may put it in the unlinked set;
1625      * it depends on whether we're the last link, and on whether there are
1626      * other holds on the vnode. So we dmdu_tx_hold() the right things to
1627      * allow for either case.
1628      */
1629     obj = zp->z_id;
1630     tx = dmdu_tx_create(zfsvfs->z_os);
1631     dmdu_tx_hold_zap(tx, dzp->z_id, FALSE, name);
1632     dmdu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
1633     zfs_sa_upgrade_txholds(tx, zp);
1634     zfs_sa_upgrade_txholds(tx, dzp);
1635     if (may_delete_now) {
1636         toobig =
1637             zp->z_size > zp->z_blkisz * DMU_MAX_DELETEBLKCNT;
1638         /* if the file is too big, only hold_free a token amount */
1639         dmdu_tx_hold_free(tx, zp->z_id, 0,
1640             (toobig ? DMU_MAX_ACCESS : DMU_OBJECT_END));
1641     }
1643     /* are there any extended attributes? */
1644     error = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1645         &xattr_obj, sizeof (xattr_obj));
1646     if (error == 0 && xattr_obj) {
1647         error = zfs_zget(zfsvfs, xattr_obj, &xzp);

```

```

1648     ASSERT0(error);
1649     ASSERT3U(error, ==, 0);
1650     dmdu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
1651     dmdu_tx_hold_sa(tx, xzp->z_sa_hdl, B_FALSE);
1652     }
1653     mutex_enter(&zp->z_lock);
1654     if ((acl_obj = zfs_external_acl(zp)) != 0 && may_delete_now)
1655         dmdu_tx_hold_free(tx, acl_obj, 0, DMU_OBJECT_END);
1656     mutex_exit(&zp->z_lock);
1658     /* charge as an update -- would be nice not to charge at all */
1659     dmdu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
1661     error = dmdu_tx_assign(tx, TXG_NOWAIT);
1662     if (error) {
1663         zfs_dirent_unlock(dl);
1664         VN_RELE(vp);
1665         if (xzp)
1666             VN_RELE(ZTOV(xzp));
1667         if (error == ERESTART) {
1668             dmdu_tx_wait(tx);
1669             dmdu_tx_abort(tx);
1670             goto top;
1671         }
1672         if (realnmp)
1673             pn_free(realnmp);
1674         dmdu_tx_abort(tx);
1675         ZFS_EXIT(zfsvfs);
1676         return (error);
1677     }
1679     /*
1680      * Remove the directory entry.
1681      */
1682     error = zfs_link_destroy(dl, zp, tx, zflg, &unlinked);
1684     if (error) {
1685         dmdu_tx_commit(tx);
1686         goto out;
1687     }
1689     if (unlinked) {
1691         /*
1692          * Hold z_lock so that we can make sure that the ACL obj
1693          * hasn't changed. Could have been deleted due to
1694          * zfs_sa_upgrade().
1695          */
1696         mutex_enter(&zp->z_lock);
1697         mutex_enter(&vp->v_lock);
1698         (void) sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
1699             &xattr_obj_unlinked, sizeof (xattr_obj_unlinked));
1700         delete_now = may_delete_now && !toobig &&
1701             vp->v_count == 1 && !vn_has_cached_data(vp) &&
1702             xattr_obj == xattr_obj_unlinked && zfs_external_acl(zp) ==
1703             acl_obj;
1704         mutex_exit(&vp->v_lock);
1705     }
1707     if (delete_now) {
1708         if (xattr_obj_unlinked) {
1709             ASSERT3U(xzp->z_links, ==, 2);
1710             mutex_enter(&xzp->z_lock);
1711             xzp->z_unlinked = 1;
1712             xzp->z_links = 0;

```

```

1713     error = sa_update(xzp->z_sa_hdl, SA_ZPL_LINKS(zfsvfs),
1714                     &xzp->z_links, sizeof (xzp->z_links), tx);
1715     ASSERT3U(error, ==, 0);
1716     mutex_exit(&xzp->z_lock);
1717     zfs_unlinked_add(xzp, tx);

1719     if (zp->z_is_sa)
1720         error = sa_remove(zp->z_sa_hdl,
1721                         SA_ZPL_XATTR(zfsvfs), tx);
1722     else
1723         error = sa_update(zp->z_sa_hdl,
1724                         SA_ZPL_XATTR(zfsvfs), &null_xattr,
1725                         sizeof (uint64_t), tx);
1726     ASSERT0(error);
1727     ASSERT3U(error, ==, 0);
1728     }
1729     mutex_enter(&vp->v_lock);
1730     vp->v_count--;
1731     ASSERT0(vp->v_count);
1732     ASSERT3U(vp->v_count, ==, 0);
1733     mutex_exit(&vp->v_lock);
1734     mutex_exit(&zp->z_lock);
1735     zfs_znode_delete(zp, tx);
1736     } else if (unlinked) {
1737     mutex_exit(&zp->z_lock);
1738     zfs_unlinked_add(zp, tx);
1739     }

1739     txtype = TX_REMOVE;
1740     if (flags & IGNORECASE)
1741         txtype |= TX_CI;
1742     zfs_log_remove(zilog, tx, txtype, dzp, name, obj);

1744     dmu_tx_commit(tx);
1745 out:
1746     if (realnmp)
1747         pn_free(realnmp);

1749     zfs_dirent_unlock(dl);

1751     if (!delete_now)
1752         VN_RELE(vp);
1753     if (xzp)
1754         VN_RELE(ZTOV(xzp));

1756     if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
1757         zil_commit(zilog, 0);

1759     ZFS_EXIT(zfsvfs);
1760     return (error);
1761 }

```

unchanged portion omitted

```

2597 /*
2598  * Set the file attributes to the values contained in the
2599  * vattr structure.
2600  *
2601  * IN:    vp      - vnode of file to be modified.
2602  *        vap     - new attribute values.
2603  *        If AT_XVATTR set, then optional attrs are being set
2604  *        flags   - ATTR_UTIME set if non-default time values provided.
2605  *        - ATTR_NOACLCHK (CIFS context only).
2606  *        cr      - credentials of caller.
2607  *        ct      - caller context
2608  *
2609  * RETURN: 0 if success

```

```

2610 *          error code if failure
2611 *
2612 * Timestamps:
2613 *   vp - ctime updated, mtime updated if size changed.
2614 */
2615 /* ARGSUSED */
2616 static int
2617 zfs_setattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
2618            caller_context_t *ct)
2619 {
2620     znode_t          *zp = VTOZ(vp);
2621     zfsvfs_t        *zfsvfs = zp->z_zfsvfs;
2622     zillog_t        *zilog;
2623     dmu_tx_t        *tx;
2624     vattr_t         oldva;
2625     tmpxvattr_t     tmpxvattr;
2626     uint_t          mask = vap->va_mask;
2627     uint_t          saved_mask;
2628     int             trim_mask = 0;
2629     uint64_t        new_mode;
2630     uint64_t        new_uid, new_gid;
2631     uint64_t        xattr_obj;
2632     uint64_t        mtime[2], ctime[2];
2633     znode_t        *attrzp;
2634     int             need_policy = FALSE;
2635     int             err, err2;
2636     zfs_fuid_info_t *fuidp = NULL;
2637     xvattr_t *xvap = (xvattr_t *)vap;      /* vap may be an xvattr_t * */
2638     xoptattr_t      *xoap;
2639     zfs_acl_t       *aclp;
2640     boolean_t       skipaclchk = (flags & ATTR_NOACLCHK) ? B_TRUE : B_FALSE;
2641     boolean_t       fuid_dirtied = B_FALSE;
2642     sa_bulk_attr_t  bulk[7], xattr_bulk[7];
2643     int             count = 0, xattr_count = 0;

2645     if (mask == 0)
2646         return (0);

2648     if (mask & AT_NOSET)
2649         return (EINVAL);

2651     ZFS_ENTER(zfsvfs);
2652     ZFS_VERIFY_ZP(zp);

2654     zilog = zfsvfs->z_log;

2656     /*
2657      * Make sure that if we have ephemeral uid/gid or xvattr specified
2658      * that file system is at proper version level
2659      */

2661     if (zfsvfs->z_use_fuids == B_FALSE &&
2662         (((mask & AT_UID) && IS_EPHEMERAL(vap->va_uid)) ||
2663          ((mask & AT_GID) && IS_EPHEMERAL(vap->va_gid)) ||
2664          (mask & AT_XVATTR))) {
2665         ZFS_EXIT(zfsvfs);
2666         return (EINVAL);
2667     }

2669     if (mask & AT_SIZE && vp->v_type == VDIR) {
2670         ZFS_EXIT(zfsvfs);
2671         return (EISDIR);
2672     }

2674     if (mask & AT_SIZE && vp->v_type != VREG && vp->v_type != VFIFO) {
2675         ZFS_EXIT(zfsvfs);

```

```

2676         return (EINVAL);
2677     }
2679     /*
2680     * If this is an xvattr_t, then get a pointer to the structure of
2681     * optional attributes. If this is NULL, then we have a vattr_t.
2682     */
2683     xoap = xva_getxoptattr(xvap);
2685     xva_init(&tmpxvattr);
2687     /*
2688     * Immutable files can only alter immutable bit and atime
2689     */
2690     if ((zp->z_pflags & ZFS_IMMUTABLE) &&
2691         ((mask & (AT_SIZE|AT_UID|AT_GID|AT_MTIME|AT_MODE)) ||
2692          ((mask & AT_XVATTR) && XVA_ISSET_REQ(xvap, XAT_CREATETIME)))) {
2693         ZFS_EXIT(zfsvfs);
2694         return (EPERM);
2695     }
2697     if ((mask & AT_SIZE) && (zp->z_pflags & ZFS_READONLY)) {
2698         ZFS_EXIT(zfsvfs);
2699         return (EPERM);
2700     }
2702     /*
2703     * Verify timestamps doesn't overflow 32 bits.
2704     * ZFS can handle large timestamps, but 32bit syscalls can't
2705     * handle times greater than 2039. This check should be removed
2706     * once large timestamps are fully supported.
2707     */
2708     if (mask & (AT_ATIME | AT_MTIME)) {
2709         if (((mask & AT_ATIME) && TIMESPEC_OVERFLOW(&vap->va_atime)) ||
2710             ((mask & AT_MTIME) && TIMESPEC_OVERFLOW(&vap->va_mtime))) {
2711             ZFS_EXIT(zfsvfs);
2712             return (Eoverflow);
2713         }
2714     }
2716 top:
2717     attrzp = NULL;
2718     aclp = NULL;
2720     /* Can this be moved to before the top label? */
2721     if (zfsvfs->z_vfs->vfs_flag & VFS_RDONLY) {
2722         ZFS_EXIT(zfsvfs);
2723         return (EROFS);
2724     }
2726     /*
2727     * First validate permissions
2728     */
2730     if (mask & AT_SIZE) {
2731         err = zfs_zaccess(zp, ACE_WRITE_DATA, 0, skipaclchk, cr);
2732         if (err) {
2733             ZFS_EXIT(zfsvfs);
2734             return (err);
2735         }
2736     }
2737     /* XXX - Note, we are not providing any open
2738     * mode flags here (like FNDELAY), so we may
2739     * block if there are locks present... this
2740     * should be addressed in openat().
2741     */

```

```

2742         /* XXX - would it be OK to generate a log record here? */
2743         err = zfs_freesp(zp, vap->va_size, 0, 0, FALSE);
2744         if (err) {
2745             ZFS_EXIT(zfsvfs);
2746             return (err);
2747         }
2748     }
2750     if (mask & (AT_ATIME|AT_MTIME) ||
2751         ((mask & AT_XVATTR) && (XVA_ISSET_REQ(xvap, XAT_HIDDEN) ||
2752          XVA_ISSET_REQ(xvap, XAT_READONLY) ||
2753          XVA_ISSET_REQ(xvap, XAT_ARCHIVE) ||
2754          XVA_ISSET_REQ(xvap, XAT_OFFLINE) ||
2755          XVA_ISSET_REQ(xvap, XAT_SPARSE) ||
2756          XVA_ISSET_REQ(xvap, XAT_CREATETIME) ||
2757          XVA_ISSET_REQ(xvap, XAT_SYSTEM)))) {
2758         need_policy = zfs_zaccess(zp, ACE_WRITE_ATTRIBUTES, 0,
2759             skipaclchk, cr);
2760     }
2762     if (mask & (AT_UID|AT_GID)) {
2763         int idmask = (mask & (AT_UID|AT_GID));
2764         int take_owner;
2765         int take_group;
2767         /*
2768         * NOTE: even if a new mode is being set,
2769         * we may clear S_ISUID/S_ISGID bits.
2770         */
2772         if (!(mask & AT_MODE))
2773             vap->va_mode = zp->z_mode;
2775         /*
2776         * Take ownership or chgrp to group we are a member of
2777         */
2779         take_owner = (mask & AT_UID) && (vap->va_uid == crgetuid(cr));
2780         take_group = (mask & AT_GID) &&
2781             zfs_groupmember(zfsvfs, vap->va_gid, cr);
2783         /*
2784         * If both AT_UID and AT_GID are set then take_owner and
2785         * take_group must both be set in order to allow taking
2786         * ownership.
2787         *
2788         * Otherwise, send the check through secpolicy_vnode_setattr()
2789         */
2792         if (((idmask == (AT_UID|AT_GID)) && take_owner && take_group) ||
2793             ((idmask == AT_UID) && take_owner) ||
2794             ((idmask == AT_GID) && take_group)) {
2795             if (zfs_zaccess(zp, ACE_WRITE_OWNER, 0,
2796                 skipaclchk, cr) == 0) {
2797                 /*
2798                 * Remove setuid/setgid for non-privileged users
2799                 */
2800                 secpolicy_setid_clear(vap, cr);
2801                 trim_mask = (mask & (AT_UID|AT_GID));
2802             } else {
2803                 need_policy = TRUE;
2804             }
2805         } else {
2806             need_policy = TRUE;
2807         }

```



```

2808     }
2810     mutex_enter(&zp->z_lock);
2811     oldva.va_mode = zp->z_mode;
2812     zfs_fuid_map_ids(zp, cr, &oldva.va_uid, &oldva.va_gid);
2813     if (mask & AT_XVATTR) {
2814         /*
2815          * Update xvattr mask to include only those attributes
2816          * that are actually changing.
2817          *
2818          * the bits will be restored prior to actually setting
2819          * the attributes so the caller thinks they were set.
2820          */
2821     if (XVA_ISSET_REQ(xvap, XAT_APPENDONLY)) {
2822         if (xoap->xoa_appendonly !=
2823             ((zp->z_pflags & ZFS_APPENDONLY) != 0)) {
2824             need_policy = TRUE;
2825         } else {
2826             XVA_CLR_REQ(xvap, XAT_APPENDONLY);
2827             XVA_SET_REQ(&tmpxvattr, XAT_APPENDONLY);
2828         }
2829     }
2831     if (XVA_ISSET_REQ(xvap, XAT_NOUNLINK)) {
2832         if (xoap->xoa_nounlink !=
2833             ((zp->z_pflags & ZFS_NOUNLINK) != 0)) {
2834             need_policy = TRUE;
2835         } else {
2836             XVA_CLR_REQ(xvap, XAT_NOUNLINK);
2837             XVA_SET_REQ(&tmpxvattr, XAT_NOUNLINK);
2838         }
2839     }
2841     if (XVA_ISSET_REQ(xvap, XAT_IMMUTABLE)) {
2842         if (xoap->xoa_immutable !=
2843             ((zp->z_pflags & ZFS_IMMUTABLE) != 0)) {
2844             need_policy = TRUE;
2845         } else {
2846             XVA_CLR_REQ(xvap, XAT_IMMUTABLE);
2847             XVA_SET_REQ(&tmpxvattr, XAT_IMMUTABLE);
2848         }
2849     }
2851     if (XVA_ISSET_REQ(xvap, XAT_NODUMP)) {
2852         if (xoap->xoa_nodump !=
2853             ((zp->z_pflags & ZFS_NODUMP) != 0)) {
2854             need_policy = TRUE;
2855         } else {
2856             XVA_CLR_REQ(xvap, XAT_NODUMP);
2857             XVA_SET_REQ(&tmpxvattr, XAT_NODUMP);
2858         }
2859     }
2861     if (XVA_ISSET_REQ(xvap, XAT_AV_MODIFIED)) {
2862         if (xoap->xoa_av_modified !=
2863             ((zp->z_pflags & ZFS_AV_MODIFIED) != 0)) {
2864             need_policy = TRUE;
2865         } else {
2866             XVA_CLR_REQ(xvap, XAT_AV_MODIFIED);
2867             XVA_SET_REQ(&tmpxvattr, XAT_AV_MODIFIED);
2868         }
2869     }
2871     if (XVA_ISSET_REQ(xvap, XAT_AV_QUARANTINED)) {
2872         if ((vp->v_type != VREG &&
2873             xoap->xoa_av_quarantined) ||

```

```

2874         xoap->xoa_av_quarantined !=
2875         ((zp->z_pflags & ZFS_AV_QUARANTINED) != 0)) {
2876             need_policy = TRUE;
2877         } else {
2878             XVA_CLR_REQ(xvap, XAT_AV_QUARANTINED);
2879             XVA_SET_REQ(&tmpxvattr, XAT_AV_QUARANTINED);
2880         }
2881     }
2883     if (XVA_ISSET_REQ(xvap, XAT_REPARSE)) {
2884         mutex_exit(&zp->z_lock);
2885         ZFS_EXIT(zfsvfs);
2886         return (EPERM);
2887     }
2889     if (need_policy == FALSE &&
2890         (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP) ||
2891          XVA_ISSET_REQ(xvap, XAT_OPAQUE))) {
2892         need_policy = TRUE;
2893     }
2894 }
2896 mutex_exit(&zp->z_lock);
2898 if (mask & AT_MODE) {
2899     if (zfs_zaccess(zp, ACE_WRITE_ACL, 0, skipaclchk, cr) == 0) {
2900         err = secpolicy_setid_setsticky_clear(vp, vap,
2901             &oldva, cr);
2902         if (err) {
2903             ZFS_EXIT(zfsvfs);
2904             return (err);
2905         }
2906         trim_mask |= AT_MODE;
2907     } else {
2908         need_policy = TRUE;
2909     }
2910 }
2912 if (need_policy) {
2913     /*
2914      * If trim_mask is set then take ownership
2915      * has been granted or write_acl is present and user
2916      * has the ability to modify mode. In that case remove
2917      * UID|GID and or MODE from mask so that
2918      * secpolicy_vnode_setattr() doesn't revoke it.
2919      */
2921     if (trim_mask) {
2922         saved_mask = vap->va_mask;
2923         vap->va_mask &= ~trim_mask;
2924     }
2925     err = secpolicy_vnode_setattr(cr, vp, vap, &oldva, flags,
2926         (int (*)(void *, int, cred_t *))zfs_zaccess_unix, zp);
2927     if (err) {
2928         ZFS_EXIT(zfsvfs);
2929         return (err);
2930     }
2932     if (trim_mask)
2933         vap->va_mask |= saved_mask;
2934 }
2936 /*
2937 * secpolicy_vnode_setattr, or take ownership may have
2938 * changed va_mask
2939 */

```

```

2940     mask = vap->va_mask;
2942     if ((mask & (AT_UID | AT_GID)) {
2943         err = sa_lookup(zp->z_sa_hdl, SA_ZPL_XATTR(zfsvfs),
2944             &xattr_obj, sizeof(xattr_obj));
2946         if (err == 0 && xattr_obj) {
2947             err = zfs_zget(zp->z_zfsvfs, xattr_obj, &attrzp);
2948             if (err)
2949                 goto out2;
2950         }
2951         if (mask & AT_UID) {
2952             new_uid = zfs_fuid_create(zfsvfs,
2953                 (uint64_t)vap->va_uid, cr, ZFS_OWNER, &fuidp);
2954             if (new_uid != zp->z_uid &&
2955                 zfs_fuid_overquota(zfsvfs, B_FALSE, new_uid)) {
2956                 if (attrzp)
2957                     VN_RELE(ZTOV(attrzp));
2958                 err = EDQUOT;
2959                 goto out2;
2960             }
2961         }
2963         if (mask & AT_GID) {
2964             new_gid = zfs_fuid_create(zfsvfs, (uint64_t)vap->va_gid,
2965                 cr, ZFS_GROUP, &fuidp);
2966             if (new_gid != zp->z_gid &&
2967                 zfs_fuid_overquota(zfsvfs, B_TRUE, new_gid)) {
2968                 if (attrzp)
2969                     VN_RELE(ZTOV(attrzp));
2970                 err = EDQUOT;
2971                 goto out2;
2972             }
2973         }
2974     }
2975     tx = dmu_tx_create(zfsvfs->z_os);
2977     if (mask & AT_MODE) {
2978         uint64_t pmode = zp->z_mode;
2979         uint64_t acl_obj;
2980         new_mode = (pmode & S_IFMT) | (vap->va_mode & ~S_IFMT);
2982         if (err = zfs_acl_chmod_setattr(zp, &aclp, new_mode))
2983             goto out;
2985         mutex_enter(&zp->z_lock);
2986         if (!zp->z_is_sa && ((acl_obj = zfs_external_acl(zp)) != 0)) {
2987             /*
2988              * Are we upgrading ACL from old V0 format
2989              * to V1 format?
2990              */
2991             if (zfsvfs->z_version >= ZPL_VERSION_FUID &&
2992                 zfs_znode_acl_version(zp) ==
2993                 ZFS_ACL_VERSION_INITIAL) {
2994                 dmu_tx_hold_free(tx, acl_obj, 0,
2995                     DMU_OBJECT_END);
2996                 dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
2997                     0, aclp->z_acl_bytes);
2998             } else {
2999                 dmu_tx_hold_write(tx, acl_obj, 0,
3000                     aclp->z_acl_bytes);
3001             }
3002         } else if (!zp->z_is_sa && aclp->z_acl_bytes > ZFS_ANCE_SPACE) {
3003             dmu_tx_hold_write(tx, DMU_NEW_OBJECT,
3004                 0, aclp->z_acl_bytes);
3005         }

```

```

3006         mutex_exit(&zp->z_lock);
3007         dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3008     } else {
3009         if ((mask & AT_XVATTR) &&
3010             XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3011             dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_TRUE);
3012         else
3013             dmu_tx_hold_sa(tx, zp->z_sa_hdl, B_FALSE);
3014     }
3016     if (attrzp) {
3017         dmu_tx_hold_sa(tx, attrzp->z_sa_hdl, B_FALSE);
3018     }
3020     fuid_dirtied = zfsvfs->z_fuid_dirty;
3021     if (fuid_dirtied)
3022         zfs_fuid_txhold(zfsvfs, tx);
3024     zfs_sa_upgrade_txholds(tx, zp);
3026     err = dmu_tx_assign(tx, TXG_NOWAIT);
3027     if (err) {
3028         if (err == ERESTART)
3029             dmu_tx_wait(tx);
3030         goto out;
3031     }
3033     count = 0;
3034     /*
3035      * Set each attribute requested.
3036      * We group settings according to the locks they need to acquire.
3037      *
3038      * Note: you cannot set ctime directly, although it will be
3039      * updated as a side-effect of calling this function.
3040      */
3043     if (mask & (AT_UID|AT_GID|AT_MODE))
3044         mutex_enter(&zp->z_acl_lock);
3045     mutex_enter(&zp->z_lock);
3047     SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_FLAGS(zfsvfs), NULL,
3048         &zp->z_pflags, sizeof(zp->z_pflags));
3050     if (attrzp) {
3051         if (mask & (AT_UID|AT_GID|AT_MODE))
3052             mutex_enter(&attrzp->z_acl_lock);
3053         mutex_enter(&attrzp->z_lock);
3054         SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3055             SA_ZPL_FLAGS(zfsvfs), NULL, &attrzp->z_pflags,
3056             sizeof(attrzp->z_pflags));
3057     }
3059     if (mask & (AT_UID|AT_GID)) {
3061         if (mask & AT_UID) {
3062             SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_UID(zfsvfs), NULL,
3063                 &new_uid, sizeof(new_uid));
3064             zp->z_uid = new_uid;
3065             if (attrzp) {
3066                 SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3067                     SA_ZPL_UID(zfsvfs), NULL, &new_uid,
3068                     sizeof(new_uid));
3069                 attrzp->z_uid = new_uid;
3070             }
3071         }

```

```

3073     if (mask & AT_GID) {
3074         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_GID(zfsvfs),
3075             NULL, &new_gid, sizeof (new_gid));
3076         zp->z_gid = new_gid;
3077         if (attrzp) {
3078             SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3079                 SA_ZPL_GID(zfsvfs), NULL, &new_gid,
3080                 sizeof (new_gid));
3081             attrzp->z_gid = new_gid;
3082         }
3083     }
3084     if (!(mask & AT_MODE)) {
3085         SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs),
3086             NULL, &new_mode, sizeof (new_mode));
3087         new_mode = zp->z_mode;
3088     }
3089     err = zfs_acl_chown_setattr(zp);
3090     ASSERT(err == 0);
3091     if (attrzp) {
3092         err = zfs_acl_chown_setattr(attrzp);
3093         ASSERT(err == 0);
3094     }
3095 }

3097 if (mask & AT_MODE) {
3098     SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MODE(zfsvfs), NULL,
3099         &new_mode, sizeof (new_mode));
3100     zp->z_mode = new_mode;
3101     ASSERT3U((uintptr_t)aclp, !=, NULL);
3102     err = zfs_aclset_common(zp, aclp, cr, tx);
3103     ASSERT0(err);
3104     ASSERT3U(err, ==, 0);
3105     if (zp->z_acl_cached)
3106         zfs_acl_free(zp->z_acl_cached);
3107     zp->z_acl_cached = aclp;
3108     aclp = NULL;
3109 }

3111 if (mask & AT_ETIME) {
3112     ZFS_TIME_ENCODE(&vap->va_etime, zp->z_etime);
3113     SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ETIME(zfsvfs), NULL,
3114         &zp->z_etime, sizeof (zp->z_etime));
3115 }

3117 if (mask & AT_MTIME) {
3118     ZFS_TIME_ENCODE(&vap->va_mtime, mtime);
3119     SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs), NULL,
3120         mtime, sizeof (mtime));
3121 }

3123 /* XXX - shouldn't this be done *before* the ATIME/MTIME checks? */
3124 if (mask & AT_SIZE && !(mask & AT_MTIME)) {
3125     SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_MTIME(zfsvfs),
3126         NULL, mtime, sizeof (mtime));
3127     SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ETIME(zfsvfs), NULL,
3128         &ctime, sizeof (ctime));
3129     zfs_tstamp_update_setup(zp, CONTENT_MODIFIED, mtime, ctime,
3130         B_TRUE);
3131 } else if (mask != 0) {
3132     SA_ADD_BULK_ATTR(bulk, count, SA_ZPL_ETIME(zfsvfs), NULL,
3133         &ctime, sizeof (ctime));
3134     zfs_tstamp_update_setup(zp, STATE_CHANGED, mtime, ctime,
3135         B_TRUE);
3136     if (attrzp) {

```

```

3137         SA_ADD_BULK_ATTR(xattr_bulk, xattr_count,
3138             SA_ZPL_ETIME(zfsvfs), NULL,
3139             &ctime, sizeof (ctime));
3140         zfs_tstamp_update_setup(attrzp, STATE_CHANGED,
3141             mtime, ctime, B_TRUE);
3142     }
3143 }
3144 /*
3145  * Do this after setting timestamps to prevent timestamp
3146  * update from toggling bit
3147  */

3149 if (xoap && (mask & AT_XVATTR)) {

3151     /*
3152     * restore trimmed off masks
3153     * so that return masks can be set for caller.
3154     */

3156     if (XVA_ISSET_REQ(&tmpxvattr, XAT_APPENDONLY)) {
3157         XVA_SET_REQ(xvap, XAT_APPENDONLY);
3158     }
3159     if (XVA_ISSET_REQ(&tmpxvattr, XAT_NOUNLINK)) {
3160         XVA_SET_REQ(xvap, XAT_NOUNLINK);
3161     }
3162     if (XVA_ISSET_REQ(&tmpxvattr, XAT_IMMUTABLE)) {
3163         XVA_SET_REQ(xvap, XAT_IMMUTABLE);
3164     }
3165     if (XVA_ISSET_REQ(&tmpxvattr, XAT_NODUMP)) {
3166         XVA_SET_REQ(xvap, XAT_NODUMP);
3167     }
3168     if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_MODIFIED)) {
3169         XVA_SET_REQ(xvap, XAT_AV_MODIFIED);
3170     }
3171     if (XVA_ISSET_REQ(&tmpxvattr, XAT_AV_QUARANTINED)) {
3172         XVA_SET_REQ(xvap, XAT_AV_QUARANTINED);
3173     }

3175     if (XVA_ISSET_REQ(xvap, XAT_AV_SCANSTAMP))
3176         ASSERT(vp->v_type == VREG);

3178     zfs_xvattr_set(zp, xvap, tx);
3179 }

3181 if (fuid_dirtied)
3182     zfs_fuid_sync(zfsvfs, tx);

3184 if (mask != 0)
3185     zfs_log_setattr(zilog, tx, TX_SETATTR, zp, vap, mask, fuidp);

3187 mutex_exit(&zp->z_lock);
3188 if (mask & (AT_UID|AT_GID|AT_MODE))
3189     mutex_exit(&zp->z_acl_lock);

3191 if (attrzp) {
3192     if (mask & (AT_UID|AT_GID|AT_MODE))
3193         mutex_exit(&attrzp->z_acl_lock);
3194     mutex_exit(&attrzp->z_lock);
3195 }
3196 out:
3197 if (err == 0 && attrzp) {
3198     err2 = sa_bulk_update(attrzp->z_sa_hdl, xattr_bulk,
3199         xattr_count, tx);
3200     ASSERT(err2 == 0);
3201 }

```

```

3203     if (attrzp)
3204         VN_RELE(ZTOV(attrzp));
3205     if (aclp)
3206         zfs_acl_free(aclp);

3208     if (fuidp) {
3209         zfs_fuid_info_free(fuidp);
3210         fuidp = NULL;
3211     }

3213     if (err) {
3214         dmu_tx_abort(tx);
3215         if (err == ERESTART)
3216             goto top;
3217     } else {
3218         err2 = sa_bulk_update(zp->z_sa_hdl, bulk, count, tx);
3219         dmu_tx_commit(tx);
3220     }

3222 out2:
3223     if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3224         zil_commit(zilog, 0);

3226     ZFS_EXIT(zfsvfs);
3227     return (err);
3228 }
_____ unchanged_portion_omitted _____

3327 /*
3328 * Move an entry from the provided source directory to the target
3329 * directory.  Change the entry name as indicated.
3330 *
3331 *     IN:     sdvp - Source directory containing the "old entry".
3332 *           snm  - Old entry name.
3333 *           tdvp - Target directory to contain the "new entry".
3334 *           tnm  - New entry name.
3335 *           cr   - credentials of caller.
3336 *           ct   - caller context
3337 *           flags - case flags
3338 *
3339 *     RETURN: 0 if success
3340 *           error code if failure
3341 *
3342 *     Timestamps:
3343 *           sdvp,tdvp - ctime|mtime updated
3344 */
3345 /*ARGSUSED*/
3346 static int
3347 zfs_rename(vnode_t *sdvp, char *snm, vnode_t *tdvp, char *tnm, cred_t *cr,
3348 caller_context_t *ct, int flags)
3349 {
3350     znode_t          *tdzp, *szp, *tzip;
3351     znode_t          *sdzp = VTOZ(sdvp);
3352     zfsvfs_t         *zfsvfs = sdzp->z_zfsvfs;
3353     zillog_t         *zillog;
3354     vnode_t          *realvp;
3355     zfs_dirlock_t    *sdl, *tdl;
3356     dmu_tx_t         *tx;
3357     zfs_zlock_t      *zl;
3358     int               cmp, serr, terr;
3359     int               error = 0;
3360     int               zflg = 0;

3362     ZFS_ENTER(zfsvfs);
3363     ZFS_VERIFY_ZP(sdzp);
3364     zillog = zfsvfs->z_log;

```

```

3366     /*
3367     * Make sure we have the real vp for the target directory.
3368     */
3369     if (VOP_REALVP(tdvp, &realvp, ct) == 0)
3370         tdvp = realvp;

3372     if (tdvp->v_vfsp != sdvp->v_vfsp || zfsctl_is_node(tdvp)) {
3373         ZFS_EXIT(zfsvfs);
3374         return (EXDEV);
3375     }

3377     tdzp = VTOZ(tdvp);
3378     ZFS_VERIFY_ZP(tdzp);
3379     if (zfsvfs->z_utf8 && u8_validate(tnm,
3380         strlen(tnm), NULL, U8_VALIDATE_ENTIRE, &error) < 0) {
3381         ZFS_EXIT(zfsvfs);
3382         return (EILSEQ);
3383     }

3385     if (flags & FIGNORECASE)
3386         zflg |= ZCLOOK;

3388 top:
3389     szp = NULL;
3390     tzip = NULL;
3391     zl = NULL;

3393     /*
3394     * This is to prevent the creation of links into attribute space
3395     * by renaming a linked file into/outof an attribute directory.
3396     * See the comment in zfs_link() for why this is considered bad.
3397     */
3398     if ((tdzp->z_pflags & ZFS_XATTR) != (sdzp->z_pflags & ZFS_XATTR)) {
3399         ZFS_EXIT(zfsvfs);
3400         return (EINVAL);
3401     }

3403     /*
3404     * Lock source and target directory entries.  To prevent deadlock,
3405     * a lock ordering must be defined.  We lock the directory with
3406     * the smallest object id first, or if it's a tie, the one with
3407     * the lexically first name.
3408     */
3409     if (sdzp->z_id < tdzp->z_id) {
3410         cmp = -1;
3411     } else if (sdzp->z_id > tdzp->z_id) {
3412         cmp = 1;
3413     } else {
3414         /*
3415         * First compare the two name arguments without
3416         * considering any case folding.
3417         */
3418         int nofold = (zfsvfs->z_norm & ~U8_TEXTPREP_TOUPPER);

3420         cmp = u8_strcmp(snm, tnm, 0, nofold, U8_UNICODE_LATEST, &error);
3421         ASSERT(error == 0 || !zfsvfs->z_utf8);
3422         if (cmp == 0) {
3423             /*
3424             * POSIX: "If the old argument and the new argument
3425             * both refer to links to the same existing file,
3426             * the rename() function shall return successfully
3427             * and perform no other action."
3428             */
3429             ZFS_EXIT(zfsvfs);
3430             return (0);

```

```

3431     }
3432     /*
3433     * If the file system is case-folding, then we may
3434     * have some more checking to do. A case-folding file
3435     * system is either supporting mixed case sensitivity
3436     * access or is completely case-insensitive. Note
3437     * that the file system is always case preserving.
3438     *
3439     * In mixed sensitivity mode case sensitive behavior
3440     * is the default. FIGIGNORECASE must be used to
3441     * explicitly request case insensitive behavior.
3442     *
3443     * If the source and target names provided differ only
3444     * by case (e.g., a request to rename 'tim' to 'Tim'),
3445     * we will treat this as a special case in the
3446     * case-insensitive mode: as long as the source name
3447     * is an exact match, we will allow this to proceed as
3448     * a name-change request.
3449     */
3450     if ((zfsvfs->z_case == ZFS_CASE_INSENSITIVE ||
3451         (zfsvfs->z_case == ZFS_CASE_MIXED &&
3452          flags & FIGIGNORECASE)) &&
3453         u8_strcmp(snm, tnm, 0, zfsvfs->z_norm, U8_UNICODE_LATEST,
3454                 &error) == 0) {
3455         /*
3456          * case preserving rename request, require exact
3457          * name matches
3458          */
3459         zflg |= ZCIEEXACT;
3460         zflg &= ~ZCILOOK;
3461     }
3462 }
3464 /*
3465 * If the source and destination directories are the same, we should
3466 * grab the z_name_lock of that directory only once.
3467 */
3468 if (sdzp == tdzp) {
3469     zflg |= ZHAVELOCK;
3470     rw_enter(&sdzp->z_name_lock, RW_READER);
3471 }
3473 if (cmp < 0) {
3474     serr = zfs_dirent_lock(&sdl, sdzp, snm, &szp,
3475                          ZEXISTS | zflg, NULL, NULL);
3476     terr = zfs_dirent_lock(&tdl,
3477                          tdzp, tnm, &tzp, ZRENAMING | zflg, NULL, NULL);
3478 } else {
3479     terr = zfs_dirent_lock(&tdl,
3480                          tdzp, tnm, &tzp, zflg, NULL, NULL);
3481     serr = zfs_dirent_lock(&sdl,
3482                          sdzp, snm, &szp, ZEXISTS | ZRENAMING | zflg,
3483                          NULL, NULL);
3484 }
3486 if (serr) {
3487     /*
3488     * Source entry invalid or not there.
3489     */
3490     if (!terr) {
3491         zfs_dirent_unlock(&tdl);
3492         if (tzp)
3493             VN_RELE(ZTOV(tzp));
3494     }
3496     if (sdzp == tdzp)

```

```

3497         rw_exit(&sdzp->z_name_lock);
3499         if (strcmp(snm, "..") == 0)
3500             serr = EINVAL;
3501         ZFS_EXIT(zfsvfs);
3502         return (serr);
3503     }
3504     if (terr) {
3505         zfs_dirent_unlock(&sdl);
3506         VN_RELE(ZTOV(szp));
3508         if (sdzp == tdzp)
3509             rw_exit(&sdzp->z_name_lock);
3511         if (strcmp(tnm, "..") == 0)
3512             terr = EINVAL;
3513         ZFS_EXIT(zfsvfs);
3514         return (terr);
3515     }
3517     /*
3518     * Must have write access at the source to remove the old entry
3519     * and write access at the target to create the new entry.
3520     * Note that if target and source are the same, this can be
3521     * done in a single check.
3522     */
3524     if (error = zfs_zaccess_rename(sdzp, szp, tdzp, tzp, cr))
3525         goto out;
3527     if (ZTOV(szp)->v_type == VDIR) {
3528         /*
3529          * Check to make sure rename is valid.
3530          * Can't do a move like this: /usr/a/b to /usr/a/b/c/d
3531          */
3532         if (error = zfs_rename_lock(szp, tdzp, sdzp, &z1))
3533             goto out;
3534     }
3536     /*
3537     * Does target exist?
3538     */
3539     if (tzp) {
3540         /*
3541          * Source and target must be the same type.
3542          */
3543         if (ZTOV(szp)->v_type == VDIR) {
3544             if (ZTOV(tzp)->v_type != VDIR) {
3545                 error = ENOTDIR;
3546                 goto out;
3547             }
3548         } else {
3549             if (ZTOV(tzp)->v_type == VDIR) {
3550                 error = EISDIR;
3551                 goto out;
3552             }
3553         }
3554     }
3555     /*
3556     * POSIX dictates that when the source and target
3557     * entries refer to the same file object, rename
3558     * must do nothing and exit without error.
3559     */
3560     if (szp->z_id == tzp->z_id) {
3561         error = 0;
3562         goto out;
3563     }

```

```

3563     }
3564
3565     vnevent_rename_src(ZTOV(szp), sdvp, snm, ct);
3566     if (tzp)
3567         vnevent_rename_dest(ZTOV(tzp), tdvp, tnm, ct);
3568
3569     /*
3570      * notify the target directory if it is not the same
3571      * as source directory.
3572      */
3573     if (tdvp != sdvp) {
3574         vnevent_rename_dest_dir(tdvp, ct);
3575     }
3576
3577     tx = dmu_tx_create(zfsvfs->z_os);
3578     dmu_tx_hold_sa(tx, szp->z_sa_hdl, B_FALSE);
3579     dmu_tx_hold_sa(tx, sdzp->z_sa_hdl, B_FALSE);
3580     dmu_tx_hold_zap(tx, sdzp->z_id, FALSE, snm);
3581     dmu_tx_hold_zap(tx, tdzp->z_id, TRUE, tnm);
3582     if (sdzp != tdzp) {
3583         dmu_tx_hold_sa(tx, tdzp->z_sa_hdl, B_FALSE);
3584         zfs_sa_upgrade_txholds(tx, tdzp);
3585     }
3586     if (tzp) {
3587         dmu_tx_hold_sa(tx, tzp->z_sa_hdl, B_FALSE);
3588         zfs_sa_upgrade_txholds(tx, tzp);
3589     }
3590
3591     zfs_sa_upgrade_txholds(tx, szp);
3592     dmu_tx_hold_zap(tx, zfsvfs->z_unlinkedobj, FALSE, NULL);
3593     error = dmu_tx_assign(tx, TXG_NOWAIT);
3594     if (error) {
3595         if (z1 != NULL)
3596             zfs_rename_unlock(&z1);
3597         zfs_dirent_unlock(sdl);
3598         zfs_dirent_unlock(tdl);
3599
3600         if (sdzp == tdzp)
3601             rw_exit(&sdzp->z_name_lock);
3602
3603         VN_RELE(ZTOV(szp));
3604         if (tzp)
3605             VN_RELE(ZTOV(tzp));
3606         if (error == ERESTART) {
3607             dmu_tx_wait(tx);
3608             dmu_tx_abort(tx);
3609             goto top;
3610         }
3611         dmu_tx_abort(tx);
3612         ZFS_EXIT(zfsvfs);
3613         return (error);
3614     }
3615
3616     if (tzp) /* Attempt to remove the existing target */
3617         error = zfs_link_destroy(tdl, tzp, tx, z1, NULL);
3618
3619     if (error == 0) {
3620         error = zfs_link_create(tdl, szp, tx, ZRENAMING);
3621         if (error == 0) {
3622             szp->z_pflags |= ZFS_AV_MODIFIED;
3623
3624             error = sa_update(szp->z_sa_hdl, SA_ZPL_FLAGS(zfsvfs),
3625                 (void *)&szp->z_pflags, sizeof (uint64_t), tx);
3626             ASSERT0(error);
3627             ASSERT3U(error, ==, 0);

```

```

3628         error = zfs_link_destroy(sdl, szp, tx, ZRENAMING, NULL);
3629         if (error == 0) {
3630             zfs_log_rename(zilog, tx, TX_RENAME |
3631                 (flags & IGNORECASE ? TX_CI : 0), sdzp,
3632                 sdl->dl_name, tdzp, tdl->dl_name, szp);
3633
3634             /*
3635              * Update path information for the target vnode
3636              */
3637             vn_renamepath(tdvp, ZTOV(szp), tnm,
3638                 strlen(tnm));
3639         } else {
3640             /*
3641              * At this point, we have successfully created
3642              * the target name, but have failed to remove
3643              * the source name. Since the create was done
3644              * with the ZRENAMING flag, there are
3645              * complications; for one, the link count is
3646              * wrong. The easiest way to deal with this
3647              * is to remove the newly created target, and
3648              * return the original error. This must
3649              * succeed; fortunately, it is very unlikely to
3650              * fail, since we just created it.
3651              */
3652             VERIFY3U(zfs_link_destroy(tdl, szp, tx,
3653                 ZRENAMING, NULL), ==, 0);
3654         }
3655     }
3656 }
3657
3658     dmu_tx_commit(tx);
3659 out:
3660     if (z1 != NULL)
3661         zfs_rename_unlock(&z1);
3662
3663     zfs_dirent_unlock(sdl);
3664     zfs_dirent_unlock(tdl);
3665
3666     if (sdzp == tdzp)
3667         rw_exit(&sdzp->z_name_lock);
3668
3669     VN_RELE(ZTOV(szp));
3670     if (tzp)
3671         VN_RELE(ZTOV(tzp));
3672
3673     if (zfsvfs->z_os->os_sync == ZFS_SYNC_ALWAYS)
3674         zil_commit(zilog, 0);
3675
3676     ZFS_EXIT(zfsvfs);
3677     return (error);
3678 }
3679 }

```

unchanged portion omitted

```

*****
52661 Thu Jul 26 14:19:32 2012
new/usr/src/uts/common/fs/zfs/zfs_znode.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 */

27 /* Portions Copyright 2007 Jeremy Teo */

29 #ifdef _KERNEL
30 #include <sys/types.h>
31 #include <sys/param.h>
32 #include <sys/time.h>
33 #include <sys/system.h>
34 #include <sys/sysmacros.h>
35 #include <sys/resource.h>
36 #include <sys/mntent.h>
37 #include <sys/mkdev.h>
38 #include <sys/u8_textprep.h>
39 #include <sys/dsl_dataset.h>
40 #include <sys/vfs.h>
41 #include <sys/vfs_opreg.h>
42 #include <sys/vnode.h>
43 #include <sys/file.h>
44 #include <sys/kmem.h>
45 #include <sys/errno.h>
46 #include <sys/unistd.h>
47 #include <sys/mode.h>
48 #include <sys/atomic.h>
49 #include <vm/pvn.h>
50 #include "fs/fs_subr.h"
51 #include <sys/zfs_dir.h>
52 #include <sys/zfs_acl.h>
53 #include <sys/zfs_ioctl.h>
54 #include <sys/zfs_rlock.h>
55 #include <sys/zfs_fuid.h>
56 #include <sys/dnode.h>
57 #include <sys/fs/zfs.h>
58 #include <sys/kidmap.h>
59 #endif /* _KERNEL */

61 #include <sys/dmu.h>

```

```

62 #include <sys/refcount.h>
63 #include <sys/stat.h>
64 #include <sys/zap.h>
65 #include <sys/zfs_znode.h>
66 #include <sys/sa.h>
67 #include <sys/zfs_sa.h>
68 #include <sys/zfs_stat.h>

70 #include "zfs_prop.h"
71 #include "zfs_comutil.h"

73 /*
74  * Define ZNODE_STATS to turn on statistic gathering. By default, it is only
75  * turned on when DEBUG is also defined.
76  */
77 #ifdef DEBUG
78 #define ZNODE_STATS
79 #endif /* DEBUG */

81 #ifdef ZNODE_STATS
82 #define ZNODE_STAT_ADD(stat)      ((stat)++)
83 #else
84 #define ZNODE_STAT_ADD(stat)      /* nothing */
85 #endif /* ZNODE_STATS */

87 /*
88  * Functions needed for userland (ie: libzpool) are not put under
89  * #ifdef _KERNEL; the rest of the functions have dependencies
90  * (such as VFS logic) that will not compile easily in userland.
91  */
92 #ifdef _KERNEL
93 /*
94  * Needed to close a small window in zfs_znode_move() that allows the zfsvfs to
95  * be freed before it can be safely accessed.
96  */
97 krwlock_t zfsvfs_lock;

99 static kmem_cache_t *znode_cache = NULL;

101 /*ARGSUSED*/
102 static void
103 znode_evict_error(dmu_buf_t *dbuf, void *user_ptr)
104 {
105     /*
106      * We should never drop all dbuf refs without first clearing
107      * the eviction callback.
108      */
109     panic("evicting znode %p\n", user_ptr);
110 }

    unchanged portion omitted

741 static uint64_t empty_xattr;
742 static uint64_t pad[4];
743 static zfs_acl_phys_t acl_phys;
744 /*
745  * Create a new DMU object to hold a zfs znode.
746  *
747  * IN:      dzp      - parent directory for new znode
748  *         vap      - file attributes for new znode
749  *         tx       - dmuf transaction id for zap operations
750  *         cr       - credentials of caller
751  *         flag     - flags:
752  *                   IS_ROOT_NODE - new object will be root
753  *                   IS_XATTR    - new object is an attribute
754  *         bonuslen - length of bonus buffer
755  *         setaclp  - File/Dir initial ACL

```

```

756 *          kuidp    - Tracks kuid allocation.
757 *
758 *      OUT:   zpp    - allocated znode
759 *
760 */
761 void
762 zfs_mknode(znode_t *dzp, vattr_t *vap, dmu_tx_t *tx, cred_t *cr,
763           uint_t flag, znode_t **zpp, zfs_acl_ids_t *acl_ids)
764 {
765     uint64_t      crtime[2], atime[2], mtime[2], ctime[2];
766     uint64_t      mode, size, links, parent, pflags;
767     uint64_t      dzp_pflags = 0;
768     uint64_t      rdev = 0;
769     zfsvfs_t      *zfsvfs = dzp->z_zfsvfs;
770     dmu_buf_t      *db;
771     timestruc_t    now;
772     uint64_t      gen, obj;
773     int            err;
774     int            bonuslen;
775     sa_handle_t    *sa_hdl;
776     dmu_object_type_t obj_type;
777     sa_bulk_attr_t sa_attrs[ZPL_END];
778     int            cnt = 0;
779     zfs_acl_locator_cb_t locate = { 0 };
780
781     ASSERT(vap && (vap->va_mask & (AT_TYPE|AT_MODE)) == (AT_TYPE|AT_MODE));
782
783     if (zfsvfs->z_replay) {
784         obj = vap->va_nodeid;
785         now = vap->va_ctime;
786         gen = vap->va_nblocks;
787     } else {
788         obj = 0;
789         gethrtime(&now);
790         gen = dmu_tx_get_txg(tx);
791     }
792
793     obj_type = zfsvfs->z_use_sa ? DMU_OT_SA : DMU_OT_ZNODE;
794     bonuslen = (obj_type == DMU_OT_SA) ?
795         DN_MAX_BONUSLEN : ZFS_OLD_ZNODE_PHYS_SIZE;
796
797     /*
798     * Create a new DMU object.
799     */
800     /*
801     * There's currently no mechanism for pre-reading the blocks that will
802     * be needed to allocate a new object, so we accept the small chance
803     * that there will be an i/o error and we will fail one of the
804     * assertions below.
805     */
806     if (vap->va_type == VDIR) {
807         if (zfsvfs->z_replay) {
808             err = zap_create_claim_norm(zfsvfs->z_os, obj,
809                                     zfsvfs->z_norm, DMU_OT_DIRECTORY_CONTENTS,
810                                     obj_type, bonuslen, tx);
811             ASSERT0(err);
812             ASSERT3U(err, ==, 0);
813         } else {
814             obj = zap_create_norm(zfsvfs->z_os,
815                                zfsvfs->z_norm, DMU_OT_DIRECTORY_CONTENTS,
816                                obj_type, bonuslen, tx);
817         }
818     } else {
819         if (zfsvfs->z_replay) {
820             err = dmu_object_claim(zfsvfs->z_os, obj,
821                                 DMU_OT_PLAIN_FILE_CONTENTS, 0,

```

```

821         obj_type, bonuslen, tx);
822         ASSERT0(err);
823         ASSERT3U(err, ==, 0);
824     } else {
825         obj = dmu_object_alloc(zfsvfs->z_os,
826                               DMU_OT_PLAIN_FILE_CONTENTS, 0,
827                               obj_type, bonuslen, tx);
828     }
829
830     ZFS_OBJ_HOLD_ENTER(zfsvfs, obj);
831     VERIFY(0 == sa_buf_hold(zfsvfs->z_os, obj, NULL, &db));
832
833     /*
834     * If this is the root, fix up the half-initialized parent pointer
835     * to reference the just-allocated physical data area.
836     */
837     if (flag & IS_ROOT_NODE) {
838         dzp->z_id = obj;
839     } else {
840         dzp_pflags = dzp->z_pflags;
841     }
842
843     /*
844     * If parent is an xattr, so am I.
845     */
846     if (dzp_pflags & ZFS_XATTR) {
847         flag |= IS_XATTR;
848     }
849
850     if (zfsvfs->z_use_fuids)
851         pflags = ZFS_ARCHIVE | ZFS_AV_MODIFIED;
852     else
853         pflags = 0;
854
855     if (vap->va_type == VDIR) {
856         size = 2;
857         links = (flag & (IS_ROOT_NODE | IS_XATTR)) ? 2 : 1;
858     } else {
859         size = links = 0;
860     }
861
862     if (vap->va_type == VBLK || vap->va_type == VCHR) {
863         rdev = zfs_expldev(vap->va_rdev);
864     }
865
866     parent = dzp->z_id;
867     mode = acl_ids->z_mode;
868     if (flag & IS_XATTR)
869         pflags |= ZFS_XATTR;
870
871     /*
872     * No execs denied will be determined when zfs_mode_compute() is called.
873     */
874     pflags |= acl_ids->z_aclp->z_hints &
875         (ZFS_ACL_TRIVIAL|ZFS_INHERIT_ACE|ZFS_ACL_AUTO_INHERIT|
876          ZFS_ACL_DEFAULTED|ZFS_ACL_PROTECTED);
877
878     ZFS_TIME_ENCODE(&now, crtime);
879     ZFS_TIME_ENCODE(&now, ctime);
880
881     if (vap->va_mask & AT_ETIME) {
882         ZFS_TIME_ENCODE(&vap->va_etime, atime);
883     } else {
884         ZFS_TIME_ENCODE(&now, atime);
885     }

```



```

887     if (vap->va_mask & AT_MTIME) {
888         ZFS_TIME_ENCODE(&vap->va_mtime, mtime);
889     } else {
890         ZFS_TIME_ENCODE(&now, mtime);
891     }
892
893     /* Now add in all of the "SA" attributes */
894     VERIFY(0 == sa_handle_get_from_db(zfsvfs->z_os, db, NULL, SA_HDL_SHARED,
895         &sa_hdl));
896
897     /*
898      * Setup the array of attributes to be replaced/set on the new file
899      * order for DMU_OT_ZNODE is critical since it needs to be constructed
900      * in the old znode_phys_t format. Don't change this ordering
901      */
902
903     if (obj_type == DMU_OT_ZNODE) {
904         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_ATIME(zfsvfs),
905             NULL, &atime, 16);
906         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_MTIME(zfsvfs),
907             NULL, &mtime, 16);
908         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_CTIME(zfsvfs),
909             NULL, &ctime, 16);
910         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_CRTIME(zfsvfs),
911             NULL, &crtime, 16);
912         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_GEN(zfsvfs),
913             NULL, &gen, 8);
914         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_MODE(zfsvfs),
915             NULL, &mode, 8);
916         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_SIZE(zfsvfs),
917             NULL, &size, 8);
918         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_PARENT(zfsvfs),
919             NULL, &parent, 8);
920     } else {
921         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_MODE(zfsvfs),
922             NULL, &mode, 8);
923         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_SIZE(zfsvfs),
924             NULL, &size, 8);
925         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_GEN(zfsvfs),
926             NULL, &gen, 8);
927         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_UID(zfsvfs), NULL,
928             &acl_ids->z_fuid, 8);
929         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_GID(zfsvfs), NULL,
930             &acl_ids->z_fgid, 8);
931         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_PARENT(zfsvfs),
932             NULL, &parent, 8);
933         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_FLAGS(zfsvfs),
934             NULL, &pflags, 8);
935         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_ATIME(zfsvfs),
936             NULL, &atime, 16);
937         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_MTIME(zfsvfs),
938             NULL, &mtime, 16);
939         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_CTIME(zfsvfs),
940             NULL, &ctime, 16);
941         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_CRTIME(zfsvfs),
942             NULL, &crtime, 16);
943     }
944
945     SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_LINKS(zfsvfs), NULL, &links, 8);
946
947     if (obj_type == DMU_OT_ZNODE) {
948         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_XATTR(zfsvfs), NULL,
949             &empty_xattr, 8);
950     }
951

```

```

952     if (obj_type == DMU_OT_ZNODE ||
953         (vap->va_type == VBLK || vap->va_type == VCHR)) {
954         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_RDEV(zfsvfs),
955             NULL, &rdev, 8);
956     }
957
958     if (obj_type == DMU_OT_ZNODE) {
959         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_FLAGS(zfsvfs),
960             NULL, &pflags, 8);
961         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_UID(zfsvfs), NULL,
962             &acl_ids->z_fuid, 8);
963         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_GID(zfsvfs), NULL,
964             &acl_ids->z_fgid, 8);
965         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_PAD(zfsvfs), NULL, pad,
966             sizeof (uint64_t) * 4);
967         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_ZNODE_ACL(zfsvfs), NULL,
968             &acl_phys, sizeof (zfs_acl_phys_t));
969     } else if (acl_ids->z_aclp->z_version >= ZFS_ACL_VERSION_FUID) {
970         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_DACL_COUNT(zfsvfs), NULL,
971             &acl_ids->z_aclp->z_acl_count, 8);
972         locate.cb_aclp = acl_ids->z_aclp;
973         SA_ADD_BULK_ATTR(sa_attrs, cnt, SA_ZPL_DACL_ACES(zfsvfs),
974             zfs_acl_data_locator, &locate,
975             acl_ids->z_aclp->z_acl_bytes);
976         mode = zfs_mode_compute(mode, acl_ids->z_aclp, &pflags,
977             acl_ids->z_fuid, acl_ids->z_fgid);
978     }
979
980     VERIFY(sa_replace_all_by_template(sa_hdl, sa_attrs, cnt, tx) == 0);
981
982     if (!(flag & IS_ROOT_NODE)) {
983         *zpp = zfs_znode_alloc(zfsvfs, db, 0, obj_type, sa_hdl);
984         ASSERT(*zpp != NULL);
985     } else {
986         /*
987          * If we are creating the root node, the "parent" we
988          * passed in is the znode for the root.
989          */
990         *zpp = dzpp;
991
992         (*zpp)->z_sa_hdl = sa_hdl;
993     }
994
995     (*zpp)->z_pflags = pflags;
996     (*zpp)->z_mode = mode;
997
998     if (vap->va_mask & AT_XVATTR)
999         zfs_xvattr_set(*zpp, (xvattr_t *)vap, tx);
1000
1001     if (obj_type == DMU_OT_ZNODE ||
1002         acl_ids->z_aclp->z_version < ZFS_ACL_VERSION_FUID) {
1003         err = zfs_aclset_common(*zpp, acl_ids->z_aclp, cr, tx);
1004         ASSERT0(err);
1005         ASSERT3P(err, ==, 0);
1006     }
1007     ZFS_OBJ_HOLD_EXIT(zfsvfs, obj);
1008 }
1009
1010 unchanged portion omitted
1011
1012 1395 /*
1013 1396 * Grow the block size for a file.
1014 1397 *
1015 1398 * IN:     zp      - znode of file to free data in.
1016 1399 *         size    - requested block size
1017 1400 *         tx      - open transaction.
1018 1401 *

```

```
1402 * NOTE: this function assumes that the znode is write locked.
1403 */
1404 void
1405 zfs_grow_blocksize(znode_t *zp, uint64_t size, dmu_tx_t *tx)
1406 {
1407     int          error;
1408     u_longlong_t dummy;
1409
1410     if (size <= zp->z_blkisz)
1411         return;
1412     /*
1413      * If the file size is already greater than the current blocksize,
1414      * we will not grow.  If there is more than one block in a file,
1415      * the blocksize cannot change.
1416      */
1417     if (zp->z_blkisz && zp->z_size > zp->z_blkisz)
1418         return;
1419
1420     error = dmu_object_set_blocksize(zp->z_zfsvfs->z_os, zp->z_id,
1421         size, 0, tx);
1422
1423     if (error == ENOTSUP)
1424         return;
1425     ASSERT0(error);
1426     ASSERT3U(error, ==, 0);
1427
1427     /* What blocksize did we actually get? */
1428     dmu_object_size_from_db(sa_get_db(zp->z_sa_hdl), &zp->z_blkisz, &dummy);
1429 }
1430
1431 unchanged portion omitted
```

```

*****
54147 Thu Jul 26 14:19:33 2012
new/usr/src/uts/common/fs/zfs/zil.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 * Copyright (c) 2011 by Delphix. All rights reserved.
25 */
26 /* Portions Copyright 2010 Robert Milkowski */

28 #include <sys/zfs_context.h>
29 #include <sys/spa.h>
30 #include <sys/dmu.h>
31 #include <sys/zap.h>
32 #include <sys/arc.h>
33 #include <sys/stat.h>
34 #include <sys/resource.h>
35 #include <sys/zil.h>
36 #include <sys/zil_impl.h>
37 #include <sys/dsl_dataset.h>
38 #include <sys/vdev_impl.h>
39 #include <sys/dmu_tx.h>
40 #include <sys/dsl_pool.h>

42 /*
43 * The zfs intent log (ZIL) saves transaction records of system calls
44 * that change the file system in memory with enough information
45 * to be able to replay them. These are stored in memory until
46 * either the DMU transaction group (txg) commits them to the stable pool
47 * and they can be discarded, or they are flushed to the stable log
48 * (also in the pool) due to a fsync, O_DSYNC or other synchronous
49 * requirement. In the event of a panic or power fail then those log
50 * records (transactions) are replayed.
51 *
52 * There is one ZIL per file system. Its on-disk (pool) format consists
53 * of 3 parts:
54 *
55 *   - ZIL header
56 *   - ZIL blocks
57 *   - ZIL records
58 *
59 * A log record holds a system call transaction. Log blocks can
60 * hold many log records and the blocks are chained together.

```

```

61 * Each ZIL block contains a block pointer (blkptr_t) to the next
62 * ZIL block in the chain. The ZIL header points to the first
63 * block in the chain. Note there is not a fixed place in the pool
64 * to hold blocks. They are dynamically allocated and freed as
65 * needed from the blocks available. Figure X shows the ZIL structure:
66 */

68 /*
69 * This global ZIL switch affects all pools
70 */
71 int zil_replay_disable = 0; /* disable intent logging replay */

73 /*
74 * Tunable parameter for debugging or performance analysis. Setting
75 * zfs_nocacheflush will cause corruption on power loss if a volatile
76 * out-of-order write cache is enabled.
77 */
78 boolean_t zfs_nocacheflush = B_FALSE;

80 static kmem_cache_t *zil_lwb_cache;

82 static void zil_async_to_sync(zilog_t *zilog, uint64_t foid);

84 #define LWB_EMPTY(lwb) ((BP_GET_LSIZE(&lwb->lwb_blk) - \
85     sizeof (zil_chain_t)) == (lwb->lwb_sz - lwb->lwb_nused))

88 /*
89 * ziltest is by and large an ugly hack, but very useful in
90 * checking replay without tedious work.
91 * When running ziltest we want to keep all itx's and so maintain
92 * a single list in the zl_itxg[] that uses a high txg: ZILTEST_TXG
93 * We subtract TXG_CONCURRENT_STATES to allow for common code.
94 */
95 #define ZILTEST_TXG (UINT64_MAX - TXG_CONCURRENT_STATES)

97 static int
98 zil_bp_compare(const void *x1, const void *x2)
99 {
100     const dva_t *dva1 = &((zil_bp_node_t *)x1)->zn_dva;
101     const dva_t *dva2 = &((zil_bp_node_t *)x2)->zn_dva;

103     if (DVA_GET_VDEV(dva1) < DVA_GET_VDEV(dva2))
104         return (-1);
105     if (DVA_GET_VDEV(dva1) > DVA_GET_VDEV(dva2))
106         return (1);

108     if (DVA_GET_OFFSET(dva1) < DVA_GET_OFFSET(dva2))
109         return (-1);
110     if (DVA_GET_OFFSET(dva1) > DVA_GET_OFFSET(dva2))
111         return (1);

113     return (0);
114 }

unchanged_portion_omitted

980 static lwb_t *
981 zil_lwb_commit(zilog_t *zilog, itx_t *itx, lwb_t *lwb)
982 {
983     lr_t *lrc = &itx->itx_lr; /* common log record */
984     lr_write_t *lrw = (lr_write_t *)lrc;
985     char *lr_buf;
986     uint64_t txg = lrc->lrc_txg;
987     uint64_t reclen = lrc->lrc_reclen;
988     uint64_t dlen = 0;

```

```

990     if (lwb == NULL)
991         return (NULL);

993     ASSERT(lwb->lwb_buf != NULL);

995     if (lrc->lrc_txtype == TX_WRITE && itx->itx_wr_state == WR_NEED_COPY)
996         dlen = P2ROUNDUP_TYPED(
997             lrw->lr_length, sizeof (uint64_t), uint64_t);

999     zillog->zl_cur_used += (reclen + dlen);

1001    zil_lwb_write_init(zilog, lwb);

1003    /*
1004     * If this record won't fit in the current log block, start a new one.
1005     */
1006    if (lwb->lwb_nused + reclen + dlen > lwb->lwb_sz) {
1007        lwb = zil_lwb_write_start(zilog, lwb);
1008        if (lwb == NULL)
1009            return (NULL);
1010        zil_lwb_write_init(zilog, lwb);
1011        ASSERT(LWB_EMPTY(lwb));
1012        if (lwb->lwb_nused + reclen + dlen > lwb->lwb_sz) {
1013            txg_wait_synced(zilog->zl_dmu_pool, txg);
1014            return (lwb);
1015        }
1016    }

1018    lr_buf = lwb->lwb_buf + lwb->lwb_nused;
1019    bcopy(lrc, lr_buf, reclen);
1020    lrc = (lr_t *)lr_buf;
1021    lrw = (lr_write_t *)lrc;

1023    /*
1024     * If it's a write, fetch the data or get its blkptr as appropriate.
1025     */
1026    if (lrc->lrc_txtype == TX_WRITE) {
1027        if (txg > spa_freeze_txg(zilog->zl_spa))
1028            txg_wait_synced(zilog->zl_dmu_pool, txg);
1029        if (itx->itx_wr_state != WR_COPIED) {
1030            char *dbuf;
1031            int error;

1033            if (dlen) {
1034                ASSERT(itx->itx_wr_state == WR_NEED_COPY);
1035                dbuf = lr_buf + reclen;
1036                lrw->lr_common.lrc_reclen += dlen;
1037            } else {
1038                ASSERT(itx->itx_wr_state == WR_INDIRECT);
1039                dbuf = NULL;
1040            }
1041            error = zillog->zl_get_data(
1042                itx->itx_private, lrw, dbuf, lwb->lwb_zio);
1043            if (error == EIO) {
1044                txg_wait_synced(zilog->zl_dmu_pool, txg);
1045                return (lwb);
1046            }
1047            if (error) {
1048                ASSERT(error == ENOENT || error == EEXIST ||
1049                    error == EALREADY);
1050                return (lwb);
1051            }
1052        }
1053    }

1055    /*

```

```

1056     * We're actually making an entry, so update lrc_seq to be the
1057     * log record sequence number. Note that this is generally not
1058     * equal to the itx sequence number because not all transactions
1059     * are synchronous, and sometimes spa_sync() gets there first.
1060     */
1061     lrc->lrc_seq = ++zillog->zl_lr_seq; /* we are single threaded */
1062     lwb->lwb_nused += reclen + dlen;
1063     lwb->lwb_max_txg = MAX(lwb->lwb_max_txg, txg);
1064     ASSERT3U(lwb->lwb_nused, <=, lwb->lwb_sz);
1065     ASSERT0(P2PHASE(lwb->lwb_nused, sizeof (uint64_t)));
1065     ASSERT3U(P2PHASE(lwb->lwb_nused, sizeof (uint64_t)), ==, 0);

1067     return (lwb);
1068 }
_____unchanged_portion_omitted_

```

```
*****
85223 Thu Jul 26 14:19:34 2012
new/usr/src/uts/common/fs/zfs/zio.c
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
_____unchanged_portion_omitted_____

2121 /*
2122 * =====
2123 * Allocate and free blocks
2124 * =====
2125 */
2126 static int
2127 zio_dva_allocate(zio_t *zio)
2128 {
2129     spa_t *spa = zio->io_spa;
2130     metaslab_class_t *mc = spa_normal_class(spa);
2131     blkptr_t *bp = zio->io_bp;
2132     int error;
2133     int flags = 0;

2135     if (zio->io_gang_leader == NULL) {
2136         ASSERT(zio->io_child_type > ZIO_CHILD_GANG);
2137         zio->io_gang_leader = zio;
2138     }

2140     ASSERT(BP_IS_HOLE(bp));
2141     ASSERT0(BP_GET_NDVAS(bp));
2141     ASSERT3U(BP_GET_NDVAS(bp), ==, 0);
2142     ASSERT3U(zio->io_prop.zp_copies, >, 0);
2143     ASSERT3U(zio->io_prop.zp_copies, <=, spa_max_replication(spa));
2144     ASSERT3U(zio->io_size, ==, BP_GET_PSIZE(bp));

2146     /*
2147     * The dump device does not support gang blocks so allocation on
2148     * behalf of the dump device (i.e. ZIO_FLAG_NODATA) must avoid
2149     * the "fast" gang feature.
2150     */
2151     flags |= (zio->io_flags & ZIO_FLAG_NODATA) ? METASLAB_GANG_AVOID : 0;
2152     flags |= (zio->io_flags & ZIO_FLAG_GANG_CHILD) ?
2153         METASLAB_GANG_CHILD : 0;
2154     error = metaslab_alloc(spa, mc, zio->io_size, bp,
2155         zio->io_prop.zp_copies, zio->io_txg, NULL, flags);

2157     if (error) {
2158         spa_dbgmsg(spa, "%s: metaslab allocation failure: zio %p, "
2159             "size %llu, error %d", spa_name(spa), zio, zio->io_size,
2160             error);
2161         if (error == ENOSPC && zio->io_size > SPA_MINBLOCKSIZE)
2162             return (zio_write_gang_block(zio));
2163         zio->io_error = error;
2164     }

2166     return (ZIO_PIPELINE_CONTINUE);
2167 }
_____unchanged_portion_omitted_____
```

```

*****
4431 Thu Jul 26 14:19:35 2012
new/usr/src/uts/common/sys/debug.h
3006 VERIFY[S,U,P] and ASSERT[S,U,P] frequently check if first argument is zero
*****
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 2010 Sun Microsystems, Inc. All rights reserved.
23 * Use is subject to license terms.
24 */

26 /*
27 * Copyright (c) 2012 by Delphix. All rights reserved.
28 */

30 /*      Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T */
31 /*      All Rights Reserved      */

33 #ifndef _SYS_DEBUG_H
34 #define _SYS_DEBUG_H

36 #include <sys/isa_defs.h>
37 #include <sys/types.h>
38 #include <sys/note.h>

40 #ifdef __cplusplus
41 extern "C" {
42 #endif

44 /*
45  * ASSERT(ex) causes a panic or debugger entry if expression ex is not
46  * true. ASSERT() is included only for debugging, and is a no-op in
47  * production kernels. VERIFY(ex), on the other hand, behaves like
48  * ASSERT and is evaluated on both debug and non-debug kernels.
49  */

51 #if defined(__STDC__)
52 extern int assfail(const char *, const char *, int);
53 #define VERIFY(EX) ((void)((EX) || assfail(#EX, __FILE__, __LINE__)))
54 #if DEBUG
55 #define ASSERT(EX) ((void)((EX) || assfail(#EX, __FILE__, __LINE__)))
56 #else
57 #define ASSERT(x) ((void)0)
58 #endif
59 #else /* defined(__STDC__) */
60 extern int assfail();
61 #define VERIFY(EX) ((void)((EX) || assfail("EX", __FILE__, __LINE__)))

```

```

62 #if DEBUG
63 #define ASSERT(EX) ((void)((EX) || assfail("EX", __FILE__, __LINE__)))
64 #else
65 #define ASSERT(x) ((void)0)
66 #endif
67 #endif /* defined(__STDC__) */

69 /*
70  * Assertion variants sensitive to the compilation data model
71  */
72 #if defined(_LP64)
73 #define ASSERT64(x)    ASSERT(x)
74 #define ASSERT32(x)
75 #else
76 #define ASSERT64(x)
77 #define ASSERT32(x)    ASSERT(x)
78 #endif

80 /*
81  * IMPLY and EQUIV are assertions of the form:
82  *
83  *     if (a) then (b)
84  * and
85  *     if (a) then (b) *AND* if (b) then (a)
86  */
87 #if DEBUG
88 #define IMPLY(A, B) \
89     ((void)((!(A)) || (B)) || \
90     assfail("( " #A " ) implies ( " #B " ), __FILE__, __LINE__))
91 #define EQUIV(A, B) \
92     ((void)((!(A) == !(B)) || \
93     assfail("( " #A " ) is equivalent to ( " #B " ), __FILE__, __LINE__))
94 #else
95 #define IMPLY(A, B) ((void)0)
96 #define EQUIV(A, B) ((void)0)
97 #endif

99 /*
100  * ASSERT3() behaves like ASSERT() except that it is an explicit conditional,
101  * and prints out the values of the left and right hand expressions as part of
102  * the panic message to ease debugging. The three variants imply the type
103  * of their arguments. ASSERT3S() is for signed data types, ASSERT3U() is
104  * for unsigned, and ASSERT3P() is for pointers. The VERIFY3*() macros
105  * have the same relationship as above.
106  */
107 extern void assfail3(const char *, uintmax_t, const char *, uintmax_t,
108     const char *, int);
109 #define VERIFY3_IMPL(LEFT, OP, RIGHT, TYPE) do { \
110     const TYPE __left = (TYPE)(LEFT); \
111     const TYPE __right = (TYPE)(RIGHT); \
112     if (!(__left OP __right)) \
113         assfail3(#LEFT " " #OP " " #RIGHT, \
114             (uintmax_t)__left, #OP, (uintmax_t)__right, \
115             __FILE__, __LINE__); \
116     _NOTE(CONSTCOND) } while (0)

118 #define VERIFY3S(x, y, z)    VERIFY3_IMPL(x, y, z, int64_t)
119 #define VERIFY3U(x, y, z)    VERIFY3_IMPL(x, y, z, uint64_t)
120 #define VERIFY3P(x, y, z)    VERIFY3_IMPL(x, y, z, uintptr_t)
121 #define VERIFY0(x)           VERIFY3_IMPL(x, ==, 0, uintmax_t)

123 #if DEBUG
124 #define ASSERT3S(x, y, z)    VERIFY3_IMPL(x, y, z, int64_t)
125 #define ASSERT3U(x, y, z)    VERIFY3_IMPL(x, y, z, uint64_t)
126 #define ASSERT3P(x, y, z)    VERIFY3_IMPL(x, y, z, uintptr_t)
127 #define ASSERT0(x)           VERIFY3_IMPL(x, ==, 0, uintmax_t)

```

```
128 #else
129 #define ASSERT3S(x, y, z)      ((void)0)
130 #define ASSERT3U(x, y, z)      ((void)0)
131 #define ASSERT3P(x, y, z)      ((void)0)
132 #define ASSERT0(x)              ((void)0)
133 #endif

135 #ifdef _KERNEL

137 extern void abort_sequence_enter(char *);
138 extern void debug_enter(char *);

140 #endif /* _KERNEL */

142 #if defined(DEBUG) && !defined(__sun)
143 /* CSTYLED */
144 #define STATIC
145 #else
146 /* CSTYLED */
147 #define STATIC static
148 #endif

150 #ifdef __cplusplus
151 }
unchanged_portion_omitted
```