```
**********************************************************
   90876 Tue Oct  1 14:04:06 2013
new/usr/src/cmd/zdb/zdb.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 563 static int
 564 verify_spacemap_refcounts(spa_t *spa)
 565 {
 566         uint64_t expected_refcount = 0;
 567         uint64_t actual_refcount;
 566         int expected_refcount, actual_refcount;

 569         (void) feature_get_refcount(spa,
 570             &spa_feature_table[SPA_FEATURE_SPACEMAP_HISTOGRAM],
 571             &expected_refcount);
 568         expected_refcount = spa_feature_get_refcount(spa,
 569             &spa_feature_table[SPA_FEATURE_SPACEMAP_HISTOGRAM]);
 572         actual_refcount = get_dtl_refcount(spa->spa_root_vdev);
 573         actual_refcount += get_metaslab_refcount(spa->spa_root_vdev);

 575         if (expected_refcount != actual_refcount) {
 576                 (void) printf("space map refcount mismatch: expected %lld != "
 577                     "actual %lld\n",
 578                     (longlong_t)expected_refcount,
 579                     (longlong_t)actual_refcount);
 574                 (void) printf("space map refcount mismatch: expected %d != "
 575                     "actual %d\n", expected_refcount, actual_refcount);
 580                 return (2);
 581         }
 582         return (0);
 583 }
______unchanged_portion_omitted_

 653 static void
 654 dump_metaslab(metaslab_t *msp)
 655 {
 656         vdev_t *vd = msp->ms_group->mg_vd;
 657         spa_t *spa = vd->vdev_spa;
 658         space_map_t *sm = msp->ms_sm;
 659         char freebuf[32];

 661         zdb_nicenum(msp->ms_size - space_map_allocated(sm), freebuf);

 663         (void) printf(
 664             "\tmetaslab %6llu   offset %12llx   spacemap %6llu   free    %5s\n",
 665             (u_longlong_t)msp->ms_id, (u_longlong_t)msp->ms_start,
 666             (u_longlong_t)space_map_object(sm), freebuf);

 668         if (dump_opt['m'] > 2 && !dump_opt['L']) {
 669                 mutex_enter(&msp->ms_lock);
 670                 metaslab_load_wait(msp);
 671                 if (!msp->ms_loaded) {
 672                         VERIFY0(metaslab_load(msp));
 673                         range_tree_stat_verify(msp->ms_tree);
 674                 }
 675                 dump_metaslab_stats(msp);
 676                 metaslab_unload(msp);
 677                 mutex_exit(&msp->ms_lock);
 678         }
```



```
 680         if (dump_opt['m'] > 1 && sm != NULL &&
 681             spa_feature_is_active(spa, SPA_FEATURE_SPACEMAP_HISTOGRAM)) {
 677             spa_feature_is_active(spa,
 678             &spa_feature_table[SPA_FEATURE_SPACEMAP_HISTOGRAM])) {
 682                 /*
 683                  * The space map histogram represents free space in chunks
 684                  * of sm_shift (i.e. bucket 0 refers to 2^sm_shift).
 685                  */
 686                 (void) printf("\tOn-disk histogram:\n");
 687                 dump_histogram(sm->sm_phys->smp_histogram,
 688                     SPACE_MAP_HISTOGRAM_SIZE(sm), sm->sm_shift);
 689         }

 691         if (dump_opt['d'] > 5 || dump_opt['m'] > 3) {
 692                 ASSERT(msp->ms_size == (1ULL << vd->vdev_ms_shift));

 694                 mutex_enter(&msp->ms_lock);
 695                 dump_spacemap(spa->spa_meta_objset, msp->ms_sm);
 696                 mutex_exit(&msp->ms_lock);
 697         }
 698 }
______unchanged_portion_omitted_

2437 static int
2438 dump_block_stats(spa_t *spa)
2439 {
2440         zdb_cb_t zcb = { 0 };
2441         zdb_blkstats_t *zb, *tzb;
2442         uint64_t norm_alloc, norm_space, total_alloc, total_found;
2443         int flags = TRAVERSE_PRE | TRAVERSE_PREFETCH_METADATA | TRAVERSE_HARD;
2444         int leaks = 0;

2446         (void) printf("\nTraversing all blocks %s%s%s%s%s...\n\n",
2447             (dump_opt['c'] || !dump_opt['L']) ? "to verify " : "",
2448             (dump_opt['c'] == 1) ? "metadata " : "",
2449             dump_opt['c'] ? "checksums " : "",
2450             (dump_opt['c'] && !dump_opt['L']) ? "and verify " : "",
2451             !dump_opt['L'] ? "nothing leaked " : "");

2453         /*
2454          * Load all space maps as SM_ALLOC maps, then traverse the pool
2455          * claiming each block we discover.  If the pool is perfectly
2456          * consistent, the space maps will be empty when we're done.
2457          * Anything left over is a leak; any block we can't claim (because
2458          * it's not part of any space map) is a double allocation,
2459          * reference to a freed block, or an unclaimed log block.
2460          */
2461         zdb_leak_init(spa, &zcb);

2463         /*
2464          * If there's a deferred-free bplist, process that first.
2465          */
2466         (void) bpobj_iterate_nofree(&spa->spa_deferred_bpobj,
2467             count_block_cb, &zcb, NULL);
2468         if (spa_version(spa) >= SPA_VERSION_DEADLISTS) {
2469                 (void) bpobj_iterate_nofree(&spa->spa_dsl_pool->dp_free_bpobj,
2470                     count_block_cb, &zcb, NULL);
2471         }
2472         if (spa_feature_is_active(spa, SPA_FEATURE_ASYNC_DESTROY)) {
2469         if (spa_feature_is_active(spa,
2470             &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
2473                 VERIFY3U(0, ==, bptree_iterate(spa->spa_meta_objset,
2474                     spa->spa_dsl_pool->dp_bptree_obj, B_FALSE, count_block_cb,
2475                     &zcb, NULL));
2476         }
```

```
2478         if (dump_opt['c'] > 1)
2479                 flags |= TRAVERSE_PREFETCH_DATA;

2481         zcb.zcb_totalasize = metaslab_class_get_alloc(spa_normal_class(spa));
2482         zcb.zcb_start = zcb.zcb_lastprint = gethrtime();
2483         zcb.zcb_haderrors |= traverse_pool(spa, 0, flags, zdb_blkptr_cb, &zcb);

2485         /*
2486          * If we've traversed the data blocks then we need to wait for those
2487          * I/Os to complete. We leverage "The Godfather" zio to wait on
2488          * all async I/Os to complete.
2489          */
2490         if (dump_opt['c']) {
2491                 (void) zio_wait(spa->spa_async_zio_root);
2492                 spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
2493                     ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE |
2494                     ZIO_FLAG_GODFATHER);
2495         }

2497         if (zcb.zcb_haderrors) {
2498                 (void) printf("\nError counts:\n\n");
2499                 (void) printf("\t%5s  %s\n", "errno", "count");
2500                 for (int e = 0; e < 256; e++) {
2501                         if (zcb.zcb_errors[e] != 0) {
2502                                 (void) printf("\t%5d  %llu\n",
2503                                     e, (u_longlong_t)zcb.zcb_errors[e]);
2504                         }
2505                 }
2506         }

2508         /*
2509          * Report any leaked segments.
2510          */
2511         zdb_leak_fini(spa);

2513         tzb = &zcb.zcb_type[ZB_TOTAL][ZDB_OT_TOTAL];

2515         norm_alloc = metaslab_class_get_alloc(spa_normal_class(spa));
2516         norm_space = metaslab_class_get_space(spa_normal_class(spa));

2518         total_alloc = norm_alloc + metaslab_class_get_alloc(spa_log_class(spa));
2519         total_found = tzb->zb_asize - zcb.zcb_dedup_asize;

2521         if (total_found == total_alloc) {
2522                 if (!dump_opt['L'])
2523                         (void) printf("\n\tNo leaks (block sum matches space"
2524                             " maps exactly)\n");
2525         } else {
2526                 (void) printf("block traversal size %llu != alloc %llu "
2527                     "(%s %lld)\n",
2528                     (u_longlong_t)total_found,
2529                     (u_longlong_t)total_alloc,
2530                     (dump_opt['L']) ? "unreachable" : "leaked",
2531                     (longlong_t)(total_alloc - total_found));
2532                 leaks = 1;
2533         }

2535         if (tzb->zb_count == 0)
2536                 return (2);

2538         (void) printf("\n");
2539         (void) printf("\tbp count:      %10llu\n",
2540             (u_longlong_t)tzb->zb_count);
2541         (void) printf("\tganged count:  %10llu\n",
2542             (longlong_t)tzb->zb_gangs);
2543         (void) printf("\tbp logical:    %10llu      avg: %6llu\n",
```



```
2544             (u_longlong_t)tzb->zb_lsize,
2545             (u_longlong_t)(tzb->zb_lsize / tzb->zb_count));
2546         (void) printf("\tbp physical:   %10llu      avg:"
2547             " %6llu     compression: %6.2f\n",
2548             (u_longlong_t)tzb->zb_psize,
2549             (u_longlong_t)(tzb->zb_psize / tzb->zb_count),
2550             (double)tzb->zb_lsize / tzb->zb_psize);
2551         (void) printf("\tbp allocated:  %10llu      avg:"
2552             " %6llu     compression: %6.2f\n",
2553             (u_longlong_t)tzb->zb_asize,
2554             (u_longlong_t)(tzb->zb_asize / tzb->zb_count),
2555             (double)tzb->zb_lsize / tzb->zb_asize);
2556         (void) printf("\tbp deduped:    %10llu    ref>1:"
2557             " %6llu   deduplication: %6.2f\n",
2558             (u_longlong_t)zcb.zcb_dedup_asize,
2559             (u_longlong_t)zcb.zcb_dedup_blocks,
2560             (double)zcb.zcb_dedup_asize / tzb->zb_asize + 1.0);
2561         (void) printf("\tSPA allocated: %10llu     used: %5.2f%%\n",
2562             (u_longlong_t)norm_alloc, 100.0 * norm_alloc / norm_space);

2564         if (tzb->zb_ditto_samevdev != 0) {
2565                 (void) printf("\tDittoed blocks on same vdev: %llu\n",
2566                     (longlong_t)tzb->zb_ditto_samevdev);
2567         }

2569         if (dump_opt['b'] >= 2) {
2570                 int l, t, level;
2571                 (void) printf("\nBlocks\tLSIZE\tPSIZE\tASIZE"
2572                     "\t  avg\t comp\t%%Total\tType\n");

2574                 for (t = 0; t <= ZDB_OT_TOTAL; t++) {
2575                         char csize[32], lsize[32], psize[32], asize[32];
2576                         char avg[32], gang[32];
2577                         char *typename;

2579                         if (t < DMU_OT_NUMTYPES)
2580                                 typename = dmu_ot[t].ot_name;
2581                         else
2582                                 typename = zdb_ot_extname[t - DMU_OT_NUMTYPES];

2584                         if (zcb.zcb_type[ZB_TOTAL][t].zb_asize == 0) {
2585                                 (void) printf("%6s\t%5s\t%5s\t%5s"
2586                                     "\t%5s\t%5s\t%6s\t%s\n",
2587                                     "-",
2588                                     "-",
2589                                     "-",
2590                                     "-",
2591                                     "-",
2592                                     "-",
2593                                     "-",
2594                                     typename);
2595                                 continue;
2596                         }

2598                         for (l = ZB_TOTAL - 1; l >= -1; l--) {
2599                                 level = (l == -1 ? ZB_TOTAL : l);
2600                                 zb = &zcb.zcb_type[level][t];

2602                                 if (zb->zb_asize == 0)
2603                                         continue;

2605                                 if (dump_opt['b'] < 3 && level != ZB_TOTAL)
2606                                         continue;

2608                                 if (level == 0 && zb->zb_asize ==
2609                                     zcb.zcb_type[ZB_TOTAL][t].zb_asize)
```

```
2610                                    continue;

2612                                zdb_nicenum(zb->zb_count, csize);
2613                                zdb_nicenum(zb->zb_lsize, lsize);
2614                                zdb_nicenum(zb->zb_psize, psize);
2615                                zdb_nicenum(zb->zb_asize, asize);
2616                                zdb_nicenum(zb->zb_asize / zb->zb_count, avg);
2617                                zdb_nicenum(zb->zb_gangs, gang);

2619                                (void) printf("%6s\t%5s\t%5s\t%5s\t%5s"
2620                                    "\t%5.2f\t%6.2f\t",
2621                                    csize, lsize, psize, asize, avg,
2622                                    (double)zb->zb_lsize / zb->zb_psize,
2623                                    100.0 * zb->zb_asize / tzb->zb_asize);

2625                                if (level == ZB_TOTAL)
2626                                        (void) printf("%s\n", typename);
2627                                else
2628                                        (void) printf("    L%d %s\n",
2629                                            level, typename);

2631                                if (dump_opt['b'] >= 3 && zb->zb_gangs > 0) {
2632                                        (void) printf("\t number of ganged "
2633                                            "blocks: %s\n", gang);
2634                                }

2636                                if (dump_opt['b'] >= 4) {
2637                                        (void) printf("psize "
2638                                            "(in 512-byte sectors): "
2639                                            "number of blocks\n");
2640                                        dump_histogram(zb->zb_psize_histogram,
2641                                            PSIZE_HISTO_SIZE, 0);
2642                                }
2643                        }
2644                }
2645        }

2647        (void) printf("\n");

2649        if (leaks)
2650                return (2);

2652        if (zcb.zcb_haderrors)
2653                return (3);

2655        return (0);
2656 }
```

*unchanged_portion_omitted_*

```
2759 static void
2760 dump_zpool(spa_t *spa)
2761 {
2762        dsl_pool_t *dp = spa_get_dsl(spa);
2763        int rc = 0;

2765        if (dump_opt['S']) {
2766                dump_simulated_ddt(spa);
2767                return;
2768        }

2770        if (!dump_opt['e'] && dump_opt['C'] > 1) {
2771                (void) printf("\nCached configuration:\n");
2772                dump_nvlist(spa->spa_config, 8);
2773        }

2775        if (dump_opt['C'])
```



```
2776                dump_config(spa);

2778        if (dump_opt['u'])
2779                dump_uberblock(&spa->spa_uberblock, "\nUberblock:\n", "\n");

2781        if (dump_opt['D'])
2782                dump_all_ddts(spa);

2784        if (dump_opt['d'] > 2 || dump_opt['m'])
2785                dump_metaslabs(spa);

2787        if (dump_opt['d'] || dump_opt['i']) {
2788                dump_dir(dp->dp_meta_objset);
2789                if (dump_opt['d'] >= 3) {
2790                        dump_bpobj(&spa->spa_deferred_bpobj,
2791                            "Deferred frees", 0);
2792                        if (spa_version(spa) >= SPA_VERSION_DEADLISTS) {
2793                                dump_bpobj(&spa->spa_dsl_pool->dp_free_bpobj,
2794                                    "Pool snapshot frees", 0);
2795                        }

2797                        if (spa_feature_is_active(spa,
2798                            SPA_FEATURE_ASYNC_DESTROY)) {
2796                            &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
2799                                dump_bptree(spa->spa_meta_objset,
2800                                    spa->spa_dsl_pool->dp_bptree_obj,
2801                                    "Pool dataset frees");
2802                        }
2803                        dump_dtl(spa->spa_root_vdev, 0);
2804                }
2805                (void) dmu_objset_find(spa_name(spa), dump_one_dir,
2806                    NULL, DS_FIND_SNAPSHOTS | DS_FIND_CHILDREN);
2807        }
2808        if (dump_opt['b'] || dump_opt['c'])
2809                rc = dump_block_stats(spa);

2811        if (rc == 0)
2812                rc = verify_spacemap_refcounts(spa);

2814        if (dump_opt['s'])
2815                show_pool_stats(spa);

2817        if (dump_opt['h'])
2818                dump_history(spa);

2820        if (rc != 0)
2821                exit(rc);
2822 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   13331 Tue Oct  1 14:04:07 2013
new/usr/src/cmd/zhack/zhack.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

285 static void
286 zhack_feature_enable_sync(void *arg, dmu_tx_t *tx)
286 feature_enable_sync(void *arg, dmu_tx_t *tx)
287 {
288         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
289         zfeature_info_t *feature = arg;

291         feature_enable_sync(spa, feature, tx);

291         spa_feature_enable(spa, feature, tx);
293         spa_history_log_internal(spa, "zhack enable feature", tx,
294             "guid=%s can_readonly=%u",
295             feature->fi_guid, feature->fi_can_readonly);
296 }

298 static void
299 zhack_do_feature_enable(int argc, char **argv)
300 {
301         char c;
302         char *desc, *target;
303         spa_t *spa;
304         objset_t *mos;
305         zfeature_info_t feature;
306         spa_feature_t nodeps[] = { SPA_FEATURE_NONE };
305         zfeature_info_t *nodeps[] = { NULL };

308         /*
309          * Features are not added to the pool's label until their refcounts
310          * are incremented, so fi_mos can just be left as false for now.
311          */
312         desc = NULL;
313         feature.fi_uname = "zhack";
314         feature.fi_mos = B_FALSE;
315         feature.fi_can_readonly = B_FALSE;
316         feature.fi_depends = nodeps;

318         optind = 1;
319         while ((c = getopt(argc, argv, "rmd:")) != -1) {
320                 switch (c) {
321                 case 'r':
322                         feature.fi_can_readonly = B_TRUE;
323                         break;
324                 case 'd':
325                         desc = strdup(optarg);
326                         break;
327                 default:
328                         usage();
329                         break;
330                 }
331         }

333         if (desc == NULL)
334                 desc = strdup("zhack injected");
335         feature.fi_desc = desc;
```



```
337         argc -= optind;
338         argv += optind;

340         if (argc < 2) {
341                 (void) fprintf(stderr, "error: missing feature or pool name\n");
342                 usage();
343         }
344         target = argv[0];
345         feature.fi_guid = argv[1];

347         if (!zfeature_is_valid_guid(feature.fi_guid))
348                 fatal(NULL, FTAG, "invalid feature guid: %s", feature.fi_guid);

350         zhack_spa_open(target, B_FALSE, FTAG, &spa);
351         mos = spa->spa_meta_objset;

353         if (zfeature_is_supported(feature.fi_guid))
352         if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
354                 fatal(spa, FTAG, "'%s' is a real feature, will not enable");
355         if (0 == zap_contains(mos, spa->spa_feat_desc_obj, feature.fi_guid))
356                 fatal(spa, FTAG, "feature already enabled: %s",
357                     feature.fi_guid);

359         VERIFY0(dsl_sync_task(spa_name(spa), NULL,
360             zhack_feature_enable_sync, &feature, 5));
359             feature_enable_sync, &feature, 5));

362         spa_close(spa, FTAG);

364         free(desc);
365 }

367 static void
368 feature_incr_sync(void *arg, dmu_tx_t *tx)
369 {
370         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
371         zfeature_info_t *feature = arg;
372         uint64_t refcount;

374         VERIFY0(feature_get_refcount(spa, feature, &refcount));
375         feature_sync(spa, feature, refcount + 1, tx);
372         spa_feature_incr(spa, feature, tx);
376         spa_history_log_internal(spa, "zhack feature incr", tx,
377             "guid=%s", feature->fi_guid);
378 }

380 static void
381 feature_decr_sync(void *arg, dmu_tx_t *tx)
382 {
383         spa_t *spa = dmu_tx_pool(tx)->dp_spa;
384         zfeature_info_t *feature = arg;
385         uint64_t refcount;

387         VERIFY0(feature_get_refcount(spa, feature, &refcount));
388         feature_sync(spa, feature, refcount - 1, tx);
383         spa_feature_decr(spa, feature, tx);
389         spa_history_log_internal(spa, "zhack feature decr", tx,
390             "guid=%s", feature->fi_guid);
391 }

393 static void
394 zhack_do_feature_ref(int argc, char **argv)
395 {
396         char c;
397         char *target;
398         boolean_t decr = B_FALSE;
```

```
399         spa_t *spa;
400         objset_t *mos;
401         zfeature_info_t feature;
402         spa_feature_t nodeps[] = { SPA_FEATURE_NONE };
397         zfeature_info_t *nodeps[] = { NULL };

404         /*
405          * fi_desc does not matter here because it was written to disk
406          * when the feature was enabled, but we need to properly set the
407          * feature for read or write based on the information we read off
408          * disk later.
409          */
410         feature.fi_uname = "zhack";
411         feature.fi_mos = B_FALSE;
412         feature.fi_desc = NULL;
413         feature.fi_depends = nodeps;

415         optind = 1;
416         while ((c = getopt(argc, argv, "md")) != -1) {
417                 switch (c) {
418                 case 'm':
419                         feature.fi_mos = B_TRUE;
420                         break;
421                 case 'd':
422                         decr = B_TRUE;
423                         break;
424                 default:
425                         usage();
426                         break;
427                 }
428         }
429         argc -= optind;
430         argv += optind;

432         if (argc < 2) {
433                 (void) fprintf(stderr, "error: missing feature or pool name\n");
434                 usage();
435         }
436         target = argv[0];
437         feature.fi_guid = argv[1];

439         if (!zfeature_is_valid_guid(feature.fi_guid))
440                 fatal(NULL, FTAG, "invalid feature guid: %s", feature.fi_guid);

442         zhack_spa_open(target, B_FALSE, FTAG, &spa);
443         mos = spa->spa_meta_objset;

445         if (zfeature_is_supported(feature.fi_guid)) {
446                 fatal(spa, FTAG,
447                     "'%s' is a real feature, will not change refcount");
448         }
440         if (0 == zfeature_lookup_guid(feature.fi_guid, NULL))
441                 fatal(spa, FTAG, "'%s' is a real feature, will not change "
442                     "refcount");

450         if (0 == zap_contains(mos, spa->spa_feat_for_read_obj,
451             feature.fi_guid)) {
452                 feature.fi_can_readonly = B_FALSE;
453         } else if (0 == zap_contains(mos, spa->spa_feat_for_write_obj,
454             feature.fi_guid)) {
455                 feature.fi_can_readonly = B_TRUE;
456         } else {
457                 fatal(spa, FTAG, "feature is not enabled: %s", feature.fi_guid);
458         }

460         if (decr) {
```



```
461                 uint64_t count;
462                 if (feature_get_refcount(spa, &feature, &count) == 0 &&
463                     count != 0) {
454         if (decr && !spa_feature_is_active(spa, &feature))
464                         fatal(spa, FTAG, "feature refcount already 0: %s",
465                             feature.fi_guid);
466                 }
467         }

469         VERIFY0(dsl_sync_task(spa_name(spa), NULL,
470             decr ? feature_decr_sync : feature_incr_sync, &feature, 5));

472         spa_close(spa, FTAG);
473 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  129948 Tue Oct  1 14:04:08 2013
new/usr/src/cmd/zpool/zpool_main.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Frederik Wessels. All rights reserved.
 * Copyright (c) 2013 by Prasad Joshi (sTec). All rights reserved.
 */

#include <assert.h>
#include <ctype.h>
#include <dirent.h>
#include <errno.h>
#include <fcntl.h>
#include <libgen.h>
#include <libintl.h>
#include <libuutil.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <unistd.h>
#include <priv.h>
#include <pwd.h>
#include <zone.h>
#include <zfs_prop.h>
#include <sys/fs/zfs.h>
#include <sys/stat.h>

#include <libzfs.h>

#include "zpool_util.h"
#include "zfs_comutil.h"
#include "zfeature_common.h"
```



```
#include "statcommon.h"

static int zpool_do_create(int, char **);
static int zpool_do_destroy(int, char **);

static int zpool_do_add(int, char **);
static int zpool_do_remove(int, char **);

static int zpool_do_list(int, char **);
static int zpool_do_iostat(int, char **);
static int zpool_do_status(int, char **);

static int zpool_do_online(int, char **);
static int zpool_do_offline(int, char **);
static int zpool_do_clear(int, char **);
static int zpool_do_reopen(int, char **);

static int zpool_do_reguid(int, char **);

static int zpool_do_attach(int, char **);
static int zpool_do_detach(int, char **);
static int zpool_do_replace(int, char **);
static int zpool_do_split(int, char **);

static int zpool_do_scrub(int, char **);

static int zpool_do_import(int, char **);
static int zpool_do_export(int, char **);

static int zpool_do_upgrade(int, char **);

static int zpool_do_history(int, char **);

static int zpool_do_get(int, char **);
static int zpool_do_set(int, char **);

/*
 * These libumem hooks provide a reasonable set of defaults for the allocator's
 * debugging facilities.
 */

#ifdef DEBUG
const char *
_umem_debug_init(void)
{
        return ("default,verbose"); /* $UMEM_DEBUG setting */
}
```

_____unchanged_portion_omitted_

```
/*
 * zpool create [-fnd] [-o property=value] ...
 *              [-O file-system-property=value] ...
 *              [-R root] [-m mountpoint] <pool> <dev> ...
 *
 *      -f      Force creation, even if devices appear in use
 *      -n      Do not create the pool, but display the resulting layout if it
 *              were to be created.
 *      -R      Create a pool under an alternate root
 *      -m      Set default mountpoint for the root dataset.  By default it's
 *              '/<pool>'
 *      -o      Set property=value.
 *      -d      Don't automatically enable all supported pool features
 *              (individual features can be enabled with -o).
 *      -O      Set fsproperty=value in the pool's root file system
 *
 * Creates the named pool according to the given vdev specification.  The
```

```
 * bulk of the vdev processing is done in get_vdev_spec() in zpool_vdev.c.  Once
 * we get the nvlist back from get_vdev_spec(), we either print out the contents
 * (if '-n' was specified), or pass it to libzfs to do the creation.
 */
int
zpool_do_create(int argc, char **argv)
{
        boolean_t force = B_FALSE;
        boolean_t dryrun = B_FALSE;
        boolean_t enable_all_pool_feat = B_TRUE;
        int c;
        nvlist_t *nvroot = NULL;
        char *poolname;
        int ret = 1;
        char *altroot = NULL;
        char *mountpoint = NULL;
        nvlist_t *fsprops = NULL;
        nvlist_t *props = NULL;
        char *propval;

        /* check options */
        while ((c = getopt(argc, argv, ":fndR:m:o:O:")) != -1) {
                switch (c) {
                case 'f':
                        force = B_TRUE;
                        break;
                case 'n':
                        dryrun = B_TRUE;
                        break;
                case 'd':
                        enable_all_pool_feat = B_FALSE;
                        break;
                case 'R':
                        altroot = optarg;
                        if (add_prop_list(zpool_prop_to_name(
                            ZPOOL_PROP_ALTROOT), optarg, &props, B_TRUE))
                                goto errout;
                        if (nvlist_lookup_string(props,
                            zpool_prop_to_name(ZPOOL_PROP_CACHEFILE),
                            &propval) == 0)
                                break;
                        if (add_prop_list(zpool_prop_to_name(
                            ZPOOL_PROP_CACHEFILE), "none", &props, B_TRUE))
                                goto errout;
                        break;
                case 'm':
                        /* Equivalent to -O mountpoint=optarg */
                        mountpoint = optarg;
                        break;
                case 'o':
                        if ((propval = strchr(optarg, '=')) == NULL) {
                                (void) fprintf(stderr, gettext("missing "
                                    "'=' for -o option\n"));
                                goto errout;
                        }
                        *propval = '\0';
                        propval++;

                        if (add_prop_list(optarg, propval, &props, B_TRUE))
                                goto errout;

                        /*
                         * If the user is creating a pool that doesn't support
                         * feature flags, don't enable any features.
                         */
                        if (zpool_name_to_prop(optarg) == ZPOOL_PROP_VERSION) {
```



```
                                char *end;
                                u_longlong_t ver;

                                ver = strtoull(propval, &end, 10);
                                if (*end == '\0' &&
                                    ver < SPA_VERSION_FEATURES) {
                                        enable_all_pool_feat = B_FALSE;
                                }
                        }
                        break;
                case 'O':
                        if ((propval = strchr(optarg, '=')) == NULL) {
                                (void) fprintf(stderr, gettext("missing "
                                    "'=' for -O option\n"));
                                goto errout;
                        }
                        *propval = '\0';
                        propval++;

                        /*
                         * Mountpoints are checked and then added later.
                         * Uniquely among properties, they can be specified
                         * more than once, to avoid conflict with -m.
                         */
                        if (0 == strcmp(optarg,
                            zfs_prop_to_name(ZFS_PROP_MOUNTPOINT))) {
                                mountpoint = propval;
                        } else if (add_prop_list(optarg, propval, &fsprops,
                            B_FALSE)) {
                                goto errout;
                        }
                        break;
                case ':':
                        (void) fprintf(stderr, gettext("missing argument for "
                            "'%c' option\n"), optopt);
                        goto badusage;
                case '?':
                        (void) fprintf(stderr, gettext("invalid option '%c'\n"),
                            optopt);
                        goto badusage;
                }
        }

        argc -= optind;
        argv += optind;

        /* get pool name and check number of arguments */
        if (argc < 1) {
                (void) fprintf(stderr, gettext("missing pool name argument\n"));
                goto badusage;
        }
        if (argc < 2) {
                (void) fprintf(stderr, gettext("missing vdev specification\n"));
                goto badusage;
        }

        poolname = argv[0];

        /*
         * As a special case, check for use of '/' in the name, and direct the
         * user to use 'zfs create' instead.
         */
        if (strchr(poolname, '/') != NULL) {
                (void) fprintf(stderr, gettext("cannot create '%s': invalid "
                    "character '/' in pool name\n"), poolname);
                (void) fprintf(stderr, gettext("use 'zfs create' to "
```

```
767                     "create a dataset\n"));
768             goto errout;
769         }

771         /* pass off to get_vdev_spec for bulk processing */
772         nvroot = make_root_vdev(NULL, force, !force, B_FALSE, dryrun,
773             argc - 1, argv + 1);
774         if (nvroot == NULL)
775             goto errout;

777         /* make_root_vdev() allows 0 toplevel children if there are spares */
778         if (!zfs_allocatable_devs(nvroot)) {
779             (void) fprintf(stderr, gettext("invalid vdev "
780                 "specification: at least one toplevel vdev must be "
781                 "specified\n"));
782             goto errout;
783         }

785         if (altroot != NULL && altroot[0] != '/') {
786             (void) fprintf(stderr, gettext("invalid alternate root '%s': "
787                 "must be an absolute path\n"), altroot);
788             goto errout;
789         }

791         /*
792          * Check the validity of the mountpoint and direct the user to use the
793          * '-m' mountpoint option if it looks like its in use.
794          */
795         if (mountpoint == NULL ||
796             (strcmp(mountpoint, ZFS_MOUNTPOINT_LEGACY) != 0 &&
797             strcmp(mountpoint, ZFS_MOUNTPOINT_NONE) != 0)) {
798             char buf[MAXPATHLEN];
799             DIR *dirp;

801             if (mountpoint && mountpoint[0] != '/') {
802                 (void) fprintf(stderr, gettext("invalid mountpoint "
803                     "'%s': must be an absolute path, 'legacy', or "
804                     "'none'\n"), mountpoint);
805                 goto errout;
806             }

808             if (mountpoint == NULL) {
809                 if (altroot != NULL)
810                     (void) snprintf(buf, sizeof (buf), "%s/%s",
811                         altroot, poolname);
812                 else
813                     (void) snprintf(buf, sizeof (buf), "/%s",
814                         poolname);
815             } else {
816                 if (altroot != NULL)
817                     (void) snprintf(buf, sizeof (buf), "%s%s",
818                         altroot, mountpoint);
819                 else
820                     (void) snprintf(buf, sizeof (buf), "%s",
821                         mountpoint);
822             }

824             if ((dirp = opendir(buf)) == NULL && errno != ENOENT) {
825                 (void) fprintf(stderr, gettext("mountpoint '%s' : "
826                     "%s\n"), buf, strerror(errno));
827                 (void) fprintf(stderr, gettext("use '-m' "
828                     "option to provide a different default\n"));
829                 goto errout;
830             } else if (dirp) {
831                 int count = 0;
```



```
833                 while (count < 3 && readdir(dirp) != NULL)
834                     count++;
835                 (void) closedir(dirp);

837                 if (count > 2) {
838                     (void) fprintf(stderr, gettext("mountpoint "
839                         "'%s' exists and is not empty\n"), buf);
840                     (void) fprintf(stderr, gettext("use '-m' "
841                         "option to provide a "
842                         "different default\n"));
843                     goto errout;
844                 }
845             }
846         }

848         /*
849          * Now that the mountpoint's validity has been checked, ensure that
850          * the property is set appropriately prior to creating the pool.
851          */
852         if (mountpoint != NULL) {
853             ret = add_prop_list(zfs_prop_to_name(ZFS_PROP_MOUNTPOINT),
854                 mountpoint, &fsprops, B_FALSE);
855             if (ret != 0)
856                 goto errout;
857         }

859         ret = 1;
860         if (dryrun) {
861             /*
862              * For a dry run invocation, print out a basic message and run
863              * through all the vdevs in the list and print out in an
864              * appropriate hierarchy.
865              */
866             (void) printf(gettext("would create '%s' with the "
867                 "following layout:\n\n"), poolname);

869             print_vdev_tree(NULL, poolname, nvroot, 0, B_FALSE);
870             if (num_logs(nvroot) > 0)
871                 print_vdev_tree(NULL, "logs", nvroot, 0, B_TRUE);

873             ret = 0;
874         } else {
875             /*
876              * Hand off to libzfs.
877              */
878             if (enable_all_pool_feat) {
879                 spa_feature_t i;
879                 int i;
880                 for (i = 0; i < SPA_FEATURES; i++) {
881                     char propname[MAXPATHLEN];
882                     zfeature_info_t *feat = &spa_feature_table[i];

884                     (void) snprintf(propname, sizeof (propname),
885                         "feature@%s", feat->fi_uname);

887                     /*
888                      * Skip feature if user specified it manually
889                      * on the command line.
890                      */
891                     if (nvlist_exists(props, propname))
892                         continue;

894                     ret = add_prop_list(propname,
895                         ZFS_FEATURE_ENABLED, &props, B_TRUE);
896                     if (ret != 0)
897                         goto errout;
```

```
898                         }
899                 }

901                 ret = 1;
902                 if (zpool_create(g_zfs, poolname,
903                     nvroot, props, fsprops) == 0) {
904                         zfs_handle_t *pool = zfs_open(g_zfs, poolname,
905                             ZFS_TYPE_FILESYSTEM);
906                         if (pool != NULL) {
907                                 if (zfs_mount(pool, NULL, 0) == 0)
908                                         ret = zfs_shareall(pool);
909                                 zfs_close(pool);
910                         }
911                 } else if (libzfs_errno(g_zfs) == EZFS_INVALIDNAME) {
912                         (void) fprintf(stderr, gettext("pool name may have "
913                             "been omitted\n"));
914                 }
915         }

917 errout:
918         nvlist_free(nvroot);
919         nvlist_free(fsprops);
920         nvlist_free(props);
921         return (ret);
922 badusage:
923         nvlist_free(fsprops);
924         nvlist_free(props);
925         usage(B_FALSE);
926         return (2);
927 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    4739 Tue Oct  1 14:04:10 2013
new/usr/src/common/zfs/zfeature_common.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  89 boolean_t
  90 zfeature_is_supported(const char *guid)
  91 {
  92         if (zfeature_checks_disable)
  93                 return (B_TRUE);

  95         for (spa_feature_t i = 0; i < SPA_FEATURES; i++) {
  95         return (0 == zfeature_lookup_guid(guid, NULL));
  96 }

  98 int
  99 zfeature_lookup_guid(const char *guid, zfeature_info_t **res)
 100 {
 101         for (int i = 0; i < SPA_FEATURES; i++) {
  96                 zfeature_info_t *feature = &spa_feature_table[i];
  97                 if (strcmp(guid, feature->fi_guid) == 0)
  98                         return (B_TRUE);
 103                 if (strcmp(guid, feature->fi_guid) == 0) {
 104                         if (res != NULL)
 105                                 *res = feature;
 106                         return (0);
  99         }
 100         return (B_FALSE);
 108         }

 110         return (ENOENT);
 101 }

 103 int
 104 zfeature_lookup_name(const char *name, spa_feature_t *res)
 114 zfeature_lookup_name(const char *name, zfeature_info_t **res)
 105 {
 106         for (spa_feature_t i = 0; i < SPA_FEATURES; i++) {
 116         for (int i = 0; i < SPA_FEATURES; i++) {
 107                 zfeature_info_t *feature = &spa_feature_table[i];
 108                 if (strcmp(name, feature->fi_uname) == 0) {
 109                         if (res != NULL)
 110                                 *res = i;
 120                                 *res = feature;
 111                         return (0);
 112                 }
 113         }

 115         return (ENOENT);
 116 }

 118 static void
 119 zfeature_register(spa_feature_t fid, const char *guid, const char *name,
 120     const char *desc, boolean_t readonly, boolean_t mos,
 121     const spa_feature_t *deps)
 129 zfeature_register(int fid, const char *guid, const char *name, const char *desc,
 130     boolean_t readonly, boolean_t mos, zfeature_info_t **deps)
 122 {
 123         zfeature_info_t *feature = &spa_feature_table[fid];
 124         static spa_feature_t nodeps[] = { SPA_FEATURE_NONE };
```



```
 133         static zfeature_info_t *nodeps[] = { NULL };

 126         ASSERT(name != NULL);
 127         ASSERT(desc != NULL);
 128         ASSERT(!readonly || !mos);
 129         ASSERT3U(fid, <, SPA_FEATURES);
 130         ASSERT(zfeature_is_valid_guid(guid));

 132         if (deps == NULL)
 133                 deps = nodeps;

 135         feature->fi_feature = fid;
 136         feature->fi_guid = guid;
 137         feature->fi_uname = name;
 138         feature->fi_desc = desc;
 139         feature->fi_can_readonly = readonly;
 140         feature->fi_mos = mos;
 141         feature->fi_depends = deps;
 142 }

 144 void
 145 zpool_feature_init(void)
 146 {
 147         zfeature_register(SPA_FEATURE_ASYNC_DESTROY,
 148             "com.delphix:async_destroy", "async_destroy",
 149             "Destroy filesystems asynchronously.", B_TRUE, B_FALSE, NULL);
 150         zfeature_register(SPA_FEATURE_EMPTY_BPOBJ,
 151             "com.delphix:empty_bpobj", "empty_bpobj",
 152             "Snapshots use less space.", B_TRUE, B_FALSE, NULL);
 153         zfeature_register(SPA_FEATURE_LZ4_COMPRESS,
 154             "org.illumos:lz4_compress", "lz4_compress",
 155             "LZ4 compression algorithm support.", B_FALSE, B_FALSE, NULL);
 156         zfeature_register(SPA_FEATURE_MULTI_VDEV_CRASH_DUMP,
 157             "com.joyent:multi_vdev_crash_dump", "multi_vdev_crash_dump",
 158             "Crash dumps to multiple vdev pools.", B_FALSE, B_FALSE, NULL);
 159         zfeature_register(SPA_FEATURE_SPACEMAP_HISTOGRAM,
 160             "com.delphix:spacemap_histogram", "spacemap_histogram",
 161             "Spacemaps maintain space histograms.", B_TRUE, B_FALSE, NULL);
 162         zfeature_register(SPA_FEATURE_EXTENSIBLE_DATASET,
 163             "com.delphix:extensible_dataset", "extensible_dataset",
 164             "Enhanced dataset functionality, used by other features.",
 165             B_FALSE, B_FALSE, NULL);
 166 }
______unchanged_portion_omitted_
```

```
**********************************************************
    2357 Tue Oct  1 14:04:14 2013
new/usr/src/common/zfs/zfeature_common.h
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

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

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

#ifndef _ZFEATURE_COMMON_H
#define _ZFEATURE_COMMON_H

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

#ifdef  __cplusplus
extern "C" {
#endif

struct zfeature_info;

typedef enum spa_feature {
        SPA_FEATURE_NONE = -1,
        SPA_FEATURE_ASYNC_DESTROY,
        SPA_FEATURE_EMPTY_BPOBJ,
        SPA_FEATURE_LZ4_COMPRESS,
        SPA_FEATURE_MULTI_VDEV_CRASH_DUMP,
        SPA_FEATURE_SPACEMAP_HISTOGRAM,
        SPA_FEATURE_EXTENSIBLE_DATASET,
        SPA_FEATURES
} spa_feature_t;

typedef struct zfeature_info {
        spa_feature_t fi_feature;
        const char *fi_uname;   /* User-facing feature name */
        const char *fi_guid;    /* On-disk feature identifier */
        const char *fi_desc;    /* Feature description */
        boolean_t fi_can_readonly; /* Can open pool readonly w/o support? */
```



```
        boolean_t fi_mos;       /* Is the feature necessary to read the MOS? */
        /* array of dependencies, terminated by SPA_FEATURE_NONE */
        const spa_feature_t *fi_depends;
        struct zfeature_info **fi_depends; /* array; null terminated */
} zfeature_info_t;

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

#define ZFS_FEATURE_DEBUG

enum spa_feature {
        SPA_FEATURE_ASYNC_DESTROY,
        SPA_FEATURE_EMPTY_BPOBJ,
        SPA_FEATURE_LZ4_COMPRESS,
        SPA_FEATURE_MULTI_VDEV_CRASH_DUMP,
        SPA_FEATURE_SPACEMAP_HISTOGRAM,
        SPA_FEATURES
} spa_feature_t;

extern zfeature_info_t spa_feature_table[SPA_FEATURES];

extern boolean_t zfeature_is_valid_guid(const char *);

extern boolean_t zfeature_is_supported(const char *);
extern int zfeature_lookup_name(const char *name, spa_feature_t *res);
extern int zfeature_lookup_guid(const char *, zfeature_info_t **res);
extern int zfeature_lookup_name(const char *, zfeature_info_t **res);

extern void zpool_feature_init(void);

#ifdef  __cplusplus
}
```

_____unchanged_portion_omitted_

```
**********************************************************
   43611 Tue Oct  1 14:04:17 2013
new/usr/src/grub/grub-0.97/stage2/fsys_zfs.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
/*
 *  GRUB  --  GRand Unified Bootloader
 *  Copyright (C) 1999,2000,2001,2002,2003,2004  Free Software Foundation, Inc.
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 */

/*
 * Copyright 2010 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
 */

/*
 * The zfs plug-in routines for GRUB are:
 *
 * zfs_mount() - locates a valid uberblock of the root pool and reads
 *               in its MOS at the memory address MOS.
 *
 * zfs_open() - locates a plain file object by following the MOS
 *              and places its dnode at the memory address DNODE.
 *
 * zfs_read() - read in the data blocks pointed by the DNODE.
 *
 * ZFS_SCRATCH is used as a working area.
 *
 * (memory addr)   MOS      DNODE       ZFS_SCRATCH
 *                  |         |           |
 *          +-------V---------V-----------V---------------+
 *   memory |       | dnode   | dnode     |  scratch      |
 *          |       | 512B    | 512B      |  area         |
 *          +-----------------------------------------------+
 */

#ifdef  FSYS_ZFS

#include "shared.h"
#include "filesys.h"
#include "fsys_zfs.h"
```



```
/* cache for a file block of the currently zfs_open()-ed file */
static void *file_buf = NULL;
static uint64_t file_start = 0;
static uint64_t file_end = 0;

/* cache for a dnode block */
static dnode_phys_t *dnode_buf = NULL;
static dnode_phys_t *dnode_mdn = NULL;
static uint64_t dnode_start = 0;
static uint64_t dnode_end = 0;

static uint64_t pool_guid = 0;
static uberblock_t current_uberblock;
static char *stackbase;

decomp_entry_t decomp_table[ZIO_COMPRESS_FUNCTIONS] =
{
        {"inherit", 0},                 /* ZIO_COMPRESS_INHERIT */
        {"on", lzjb_decompress},        /* ZIO_COMPRESS_ON */
        {"off", 0},                     /* ZIO_COMPRESS_OFF */
        {"lzjb", lzjb_decompress},      /* ZIO_COMPRESS_LZJB */
        {"empty", 0},                   /* ZIO_COMPRESS_EMPTY */
        {"gzip-1", 0},                  /* ZIO_COMPRESS_GZIP_1 */
        {"gzip-2", 0},                  /* ZIO_COMPRESS_GZIP_2 */
        {"gzip-3", 0},                  /* ZIO_COMPRESS_GZIP_3 */
        {"gzip-4", 0},                  /* ZIO_COMPRESS_GZIP_4 */
        {"gzip-5", 0},                  /* ZIO_COMPRESS_GZIP_5 */
        {"gzip-6", 0},                  /* ZIO_COMPRESS_GZIP_6 */
        {"gzip-7", 0},                  /* ZIO_COMPRESS_GZIP_7 */
        {"gzip-8", 0},                  /* ZIO_COMPRESS_GZIP_8 */
        {"gzip-9", 0},                  /* ZIO_COMPRESS_GZIP_9 */
        {"zle", 0},                     /* ZIO_COMPRESS_ZLE */
        {"lz4", lz4_decompress}         /* ZIO_COMPRESS_LZ4 */
};
_____unchanged_portion_omitted_

/*
 * List of pool features that the grub implementation of ZFS supports for
 * read. Note that features that are only required for write do not need
 * to be listed here since grub opens pools in read-only mode.
 *
 * When this list is updated the version number in usr/src/grub/capability
 * must be incremented to ensure the new grub gets installed.
 */
static const char *spa_feature_names[] = {
        "org.illumos:lz4_compress",
        "com.delphix:extensible_dataset",
        NULL
};
_____unchanged_portion_omitted_

/*
 * Given a MOS metadnode, get the metadnode of a given filesystem name (fsname),
 * e.g. pool/rootfs, or a given object number (obj), e.g. the object number
 * of pool/rootfs.
 *
 * If no fsname and no obj are given, return the DSL_DIR metadnode.
 * If fsname is given, return its metadnode and its matching object number.
 * If only obj is given, return the metadnode for this object number.
 *
 * Return:
 *      0 - success
 *      errnum - failure
 */
static int
get_objset_mdn(dnode_phys_t *mosmdn, char *fsname, uint64_t *obj,
```

```
1026     dnode_phys_t *mdn, char *stack)
1027 {
1028         uint64_t objnum, headobj;
1029         char *cname, ch;
1030         blkptr_t *bp;
1031         objset_phys_t *osp;
1032         int issnapshot = 0;
1033         char *snapname;

1035         if (fsname == NULL && obj) {
1036                 headobj = *obj;
1037                 goto skip;
1038         }

1040         if (errnum = dnode_get(mosmdn, DMU_POOL_DIRECTORY_OBJECT,
1041             DMU_OT_OBJECT_DIRECTORY, mdn, stack))
1042                 return (errnum);

1044         if (errnum = zap_lookup(mdn, DMU_POOL_ROOT_DATASET, &objnum,
1045             stack))
1046                 return (errnum);

1048         if (errnum = dnode_get(mosmdn, objnum, 0, mdn, stack))
1047         if (errnum = dnode_get(mosmdn, objnum, DMU_OT_DSL_DIR, mdn, stack))
1049                 return (errnum);

1051         if (fsname == NULL) {
1052                 headobj =
1053                     ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_head_dataset_obj;
1054                 goto skip;
1055         }

1057         /* take out the pool name */
1058         while (*fsname && !grub_isspace(*fsname) && *fsname != '/')
1059                 fsname++;

1061         while (*fsname && !grub_isspace(*fsname)) {
1062                 uint64_t childobj;

1064                 while (*fsname == '/')
1065                         fsname++;

1067                 cname = fsname;
1068                 while (*fsname && !grub_isspace(*fsname) && *fsname != '/')
1069                         fsname++;
1070                 ch = *fsname;
1071                 *fsname = 0;

1073                 snapname = cname;
1074                 while (*snapname && !grub_isspace(*snapname) && *snapname !=
1075                     '@')
1076                         snapname++;
1077                 if (*snapname == '@') {
1078                         issnapshot = 1;
1079                         *snapname = 0;
1080                 }
1081                 childobj =
1082                     ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_child_dir_zapobj;
1083                 if (errnum = dnode_get(mosmdn, childobj,
1084                     DMU_OT_DSL_DIR_CHILD_MAP, mdn, stack))
1085                         return (errnum);

1087                 if (zap_lookup(mdn, cname, &objnum, stack))
1088                         return (ERR_FILESYSTEM_NOT_FOUND);

1090                 if (errnum = dnode_get(mosmdn, objnum, 0,
```



```
1089                 if (errnum = dnode_get(mosmdn, objnum, DMU_OT_DSL_DIR,
1091                     mdn, stack))
1092                         return (errnum);

1094                 *fsname = ch;
1095                 if (issnapshot)
1096                         *snapname = '@';
1097         }
1098         headobj = ((dsl_dir_phys_t *)DN_BONUS(mdn))->dd_head_dataset_obj;
1099         if (obj)
1100                 *obj = headobj;

1102 skip:
1103         if (errnum = dnode_get(mosmdn, headobj, 0, mdn, stack))
1102         if (errnum = dnode_get(mosmdn, headobj, DMU_OT_DSL_DATASET, mdn, stack))
1104                 return (errnum);
1105         if (issnapshot) {
1106                 uint64_t snapobj;

1108                 snapobj = ((dsl_dataset_phys_t *)DN_BONUS(mdn))->
1109                     ds_snapnames_zapobj;

1111                 if (errnum = dnode_get(mosmdn, snapobj,
1112                     DMU_OT_DSL_DS_SNAP_MAP, mdn, stack))
1113                         return (errnum);
1114                 if (zap_lookup(mdn, snapname + 1, &headobj, stack))
1115                         return (ERR_FILESYSTEM_NOT_FOUND);
1116                 if (errnum = dnode_get(mosmdn, headobj, 0, mdn, stack))
1115                 if (errnum = dnode_get(mosmdn, headobj,
1116                     DMU_OT_DSL_DATASET, mdn, stack))
1117                         return (errnum);
1118                 if (obj)
1119                         *obj = headobj;
1120         }

1122         bp = &((dsl_dataset_phys_t *)DN_BONUS(mdn))->ds_bp;
1123         osp = (objset_phys_t *)stack;
1124         stack += sizeof (objset_phys_t);
1125         if (errnum = zio_read(bp, osp, stack))
1126                 return (errnum);

1128         grub_memmove((char *)mdn, (char *)&osp->os_meta_dnode, DNODE_SIZE);

1130         return (0);
1131 }
```

*_____unchanged_portion_omitted_*

```
**********************************************************
  102954 Tue Oct  1 14:04:19 2013
new/usr/src/lib/libzfs/common/libzfs_pool.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright 2011 Nexenta Systems, Inc. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
 */

#include <ctype.h>
#include <errno.h>
#include <devid.h>
#include <fcntl.h>
#include <libintl.h>
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <libgen.h>
#include <sys/efi_partition.h>
#include <sys/vtoc.h>
#include <sys/zfs_ioctl.h>
#include <dlfcn.h>

#include "zfs_namecheck.h"
#include "zfs_prop.h"
#include "libzfs_impl.h"
#include "zfs_comutil.h"
#include "zfeature_common.h"

static int read_efi_label(nvlist_t *config, diskaddr_t *sb);

#define DISK_ROOT       "/dev/dsk"
#define RDISK_ROOT      "/dev/rdsk"
#define BACKUP_SLICE    "s2"

typedef struct prop_flags {
```



```
        int create:1;   /* Validate property on creation */
        int import:1;   /* Validate property on import */
} prop_flags_t;
______unchanged_portion_omitted_


/*
 * Given an nvlist of zpool properties to be set, validate that they are
 * correct, and parse any numeric properties (index, boolean, etc) if they are
 * specified as strings.
 */
static nvlist_t *
zpool_valid_proplist(libzfs_handle_t *hdl, const char *poolname,
    nvlist_t *props, uint64_t version, prop_flags_t flags, char *errbuf)
{
        nvpair_t *elem;
        nvlist_t *retprops;
        zpool_prop_t prop;
        char *strval;
        uint64_t intval;
        char *slash, *check;
        struct stat64 statbuf;
        zpool_handle_t *zhp;
        nvlist_t *nvroot;

        if (nvlist_alloc(&retprops, NV_UNIQUE_NAME, 0) != 0) {
                (void) no_memory(hdl);
                return (NULL);
        }

        elem = NULL;
        while ((elem = nvlist_next_nvpair(props, elem)) != NULL) {
                const char *propname = nvpair_name(elem);

                prop = zpool_name_to_prop(propname);
                if (prop == ZPROP_INVAL && zpool_prop_feature(propname)) {
                        int err;
                        zfeature_info_t *feature;
                        char *fname = strchr(propname, '@') + 1;

                        err = zfeature_lookup_name(fname, NULL);
                        err = zfeature_lookup_name(fname, &feature);
                        if (err != 0) {
                                ASSERT3U(err, ==, ENOENT);
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "invalid feature '%s'"), fname);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }

                        if (nvpair_type(elem) != DATA_TYPE_STRING) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "'%s' must be a string"), propname);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }

                        (void) nvpair_value_string(elem, &strval);
                        if (strcmp(strval, ZFS_FEATURE_ENABLED) != 0) {
                                zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
                                    "property '%s' can only be set to "
                                    "'enabled'"), propname);
                                (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
                                goto error;
                        }
```

```
443                         if (nvlist_add_uint64(retprops, propname, 0) != 0) {
444                                 (void) no_memory(hdl);
445                                 goto error;
446                         }
447                         continue;
448                 }

450                 /*
451                  * Make sure this property is valid and applies to this type.
452                  */
453                 if (prop == ZPROP_INVAL) {
454                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
455                             "invalid property '%s'"), propname);
456                         (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
457                         goto error;
458                 }

460                 if (zpool_prop_readonly(prop)) {
461                         zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "'%s' "
462                             "is readonly"), propname);
463                         (void) zfs_error(hdl, EZFS_PROPREADONLY, errbuf);
464                         goto error;
465                 }

467                 if (zprop_parse_value(hdl, elem, prop, ZFS_TYPE_POOL, retprops,
468                     &strval, &intval, errbuf) != 0)
469                         goto error;

471                 /*
472                  * Perform additional checking for specific properties.
473                  */
474                 switch (prop) {
475                 case ZPOOL_PROP_VERSION:
476                         if (intval < version ||
477                             !SPA_VERSION_IS_SUPPORTED(intval)) {
478                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
479                                     "property '%s' number %d is invalid."),
480                                     propname, intval);
481                                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
482                                 goto error;
483                         }
484                         break;

486                 case ZPOOL_PROP_BOOTFS:
487                         if (flags.create || flags.import) {
488                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
489                                     "property '%s' cannot be set at creation "
490                                     "or import time"), propname);
491                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
492                                 goto error;
493                         }

495                         if (version < SPA_VERSION_BOOTFS) {
496                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
497                                     "pool must be upgraded to support "
498                                     "'%s' property"), propname);
499                                 (void) zfs_error(hdl, EZFS_BADVERSION, errbuf);
500                                 goto error;
501                         }

503                         /*
504                          * bootfs property value has to be a dataset name and
505                          * the dataset has to be in the same pool as it sets to.
506                          */
507                         if (strval[0] != '\0' && !bootfs_name_valid(poolname,
508                             strval)) {
```



```
509                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN, "'%s' "
510                                     "is an invalid name"), strval);
511                                 (void) zfs_error(hdl, EZFS_INVALIDNAME, errbuf);
512                                 goto error;
513                         }

515                         if ((zhp = zpool_open_canfail(hdl, poolname)) == NULL) {
516                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
517                                     "could not open pool '%s'"), poolname);
518                                 (void) zfs_error(hdl, EZFS_OPENFAILED, errbuf);
519                                 goto error;
520                         }
521                         verify(nvlist_lookup_nvlist(zpool_get_config(zhp, NULL),
522                             ZPOOL_CONFIG_VDEV_TREE, &nvroot) == 0);

524                         /*
525                          * bootfs property cannot be set on a disk which has
526                          * been EFI labeled.
527                          */
528                         if (pool_uses_efi(nvroot)) {
529                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
530                                     "property '%s' not supported on "
531                                     "EFI labeled devices"), propname);
532                                 (void) zfs_error(hdl, EZFS_POOL_NOTSUP, errbuf);
533                                 zpool_close(zhp);
534                                 goto error;
535                         }
536                         zpool_close(zhp);
537                         break;

539                 case ZPOOL_PROP_ALTROOT:
540                         if (!flags.create && !flags.import) {
541                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
542                                     "property '%s' can only be set during pool "
543                                     "creation or import"), propname);
544                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
545                                 goto error;
546                         }

548                         if (strval[0] != '/') {
549                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
550                                     "bad alternate root '%s'"), strval);
551                                 (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
552                                 goto error;
553                         }
554                         break;

556                 case ZPOOL_PROP_CACHEFILE:
557                         if (strval[0] == '\0')
558                                 break;

560                         if (strcmp(strval, "none") == 0)
561                                 break;

563                         if (strval[0] != '/') {
564                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
565                                     "property '%s' must be empty, an "
566                                     "absolute path, or 'none'"), propname);
567                                 (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
568                                 goto error;
569                         }

571                         slash = strrchr(strval, '/');

573                         if (slash[1] == '\0' || strcmp(slash, "/.") == 0 ||
574                             strcmp(slash, "/..") == 0) {
```

```
575                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
576                                     "'%s' is not a valid file"), strval);
577                                 (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
578                                 goto error;
579                         }

581                         *slash = '\0';

583                         if (strval[0] != '\0' &&
584                             (stat64(strval, &statbuf) != 0 ||
585                             !S_ISDIR(statbuf.st_mode))) {
586                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
587                                     "'%s' is not a valid directory"),
588                                     strval);
589                                 (void) zfs_error(hdl, EZFS_BADPATH, errbuf);
590                                 goto error;
591                         }

593                         *slash = '/';
594                         break;

596                 case ZPOOL_PROP_COMMENT:
597                         for (check = strval; *check != '\0'; check++) {
598                                 if (!isprint(*check)) {
599                                         zfs_error_aux(hdl,
600                                             dgettext(TEXT_DOMAIN,
601                                             "comment may only have printable "
602                                             "characters"));
603                                         (void) zfs_error(hdl, EZFS_BADPROP,
604                                             errbuf);
605                                         goto error;
606                                 }
607                         }
608                         if (strlen(strval) > ZPROP_MAX_COMMENT) {
609                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
610                                     "comment must not exceed %d characters"),
611                                     ZPROP_MAX_COMMENT);
612                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
613                                 goto error;
614                         }
615                         break;
616                 case ZPOOL_PROP_READONLY:
617                         if (!flags.import) {
618                                 zfs_error_aux(hdl, dgettext(TEXT_DOMAIN,
619                                     "property '%s' can only be set at "
620                                     "import time"), propname);
621                                 (void) zfs_error(hdl, EZFS_BADPROP, errbuf);
622                                 goto error;
623                         }
624                         break;
625                 }
626         }

628         return (retprops);
629 error:
630         nvlist_free(retprops);
631         return (NULL);
632 }
```

*unchanged_portion_omitted_*

```
786 /*
787  * Get the state for the given feature on the given ZFS pool.
788  */
789 int
790 zpool_prop_get_feature(zpool_handle_t *zhp, const char *propname, char *buf,
791     size_t len)
```



```
792 {
793         uint64_t refcount;
794         boolean_t found = B_FALSE;
795         nvlist_t *features = zpool_get_features(zhp);
796         boolean_t supported;
797         const char *feature = strchr(propname, '@') + 1;

799         supported = zpool_prop_feature(propname);
800         ASSERT(supported || zfs_prop_unsupported(propname));

802         /*
803          * Convert from feature name to feature guid. This conversion is
804          * unecessary for unsupported@... properties because they already
805          * use guids.
806          */
807         if (supported) {
808                 int ret;
809                 spa_feature_t fid;
810                 zfeature_info_t *fi;

811                 ret = zfeature_lookup_name(feature, &fid);
812                 ret = zfeature_lookup_name(feature, &fi);
812                 if (ret != 0) {
813                         (void) strlcpy(buf, "-", len);
814                         return (ENOTSUP);
815                 }
816                 feature = spa_feature_table[fid].fi_guid;
817                 feature = fi->fi_guid;
817         }

819         if (nvlist_lookup_uint64(features, feature, &refcount) == 0)
820                 found = B_TRUE;

822         if (supported) {
823                 if (!found) {
824                         (void) strlcpy(buf, ZFS_FEATURE_DISABLED, len);
825                 } else  {
826                         if (refcount == 0)
827                                 (void) strlcpy(buf, ZFS_FEATURE_ENABLED, len);
828                         else
829                                 (void) strlcpy(buf, ZFS_FEATURE_ACTIVE, len);
830                 }
831         } else {
832                 if (found) {
833                         if (refcount == 0) {
834                                 (void) strcpy(buf, ZFS_UNSUPPORTED_INACTIVE);
835                         } else {
836                                 (void) strcpy(buf, ZFS_UNSUPPORTED_READONLY);
837                         }
838                 } else {
839                         (void) strlcpy(buf, "-", len);
840                         return (ENOTSUP);
841                 }
842         }

844         return (0);
845 }
```

*unchanged_portion_omitted_*

```
**********************************************************
    9655 Tue Oct  1 14:04:22 2013
new/usr/src/man/man5/zpool-features.5
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
'\" te
.\" Copyright (c) 2013 by Delphix. All rights reserved.
.\" Copyright (c) 2013 by Saso Kiselkov. All rights reserved.
.\" Copyright (c) 2013, Joyent, Inc. All rights reserved.
.\" The contents of this file are subject to the terms of the Common Development
.\" and Distribution License (the "License").  You may not use this file except
.\" in compliance with the License. You can obtain a copy of the license at
.\" usr/src/OPENSOLARIS.LICENSE or http://www.opensolaris.org/os/licensing.
.\"
.\" See the License for the specific language governing permissions and
.\" limitations under the License. When distributing Covered Code, include this
.\" CDDL HEADER in each file and include the License file at
.\" usr/src/OPENSOLARIS.LICENSE.  If applicable, add the following below this
.\" CDDL HEADER, with the fields enclosed by brackets "[]" replaced with your
.\" own identifying information:
.\" Portions Copyright [yyyy] [name of copyright owner]
.TH ZPOOL-FEATURES 5 "Aug 27, 2013"
.SH NAME
zpool\-features \- ZFS pool feature descriptions
.SH DESCRIPTION
.sp
.LP
ZFS pool on\-disk format versions are specified via "features" which replace
the old on\-disk format numbers (the last supported on\-disk format number is
28). To enable a feature on a pool use the \fBupgrade\fR subcommand of the
\fBzpool\fR(1M) command, or set the \fBfeature@\fR\fIfeature_name\fR property
to \fBenabled\fR.
.sp
.LP
The pool format does not affect file system version compatibility or the ability
to send file systems between pools.
.sp
.LP
Since most features can be enabled independently of each other the on\-disk
format of the pool is specified by the set of all features marked as
\fBactive\fR on the pool. If the pool was created by another software version
this set may include unsupported features.
.SS "Identifying features"
.sp
.LP
Every feature has a guid of the form \fIcom.example:feature_name\fR. The reverse
DNS name ensures that the feature's guid is unique across all ZFS
implementations. When unsupported features are encountered on a pool they will
be identified by their guids. Refer to the documentation for the ZFS
implementation that created the pool for information about those features.
.sp
.LP
Each supported feature also has a short name. By convention a feature's short
name is the portion of its guid which follows the ':' (e.g.
\fIcom.example:feature_name\fR would have the short name \fIfeature_name\fR),
however a feature's short name may differ across ZFS implementations if
following the convention would result in name conflicts.
.SS "Feature states"
.sp
.LP
Features can be in one of three states:
.sp
```



```
.ne 2
.na
\fB\fBactive\fR\fR
.ad
.RS 12n
This feature's on\-disk format changes are in effect on the pool. Support for
this feature is required to import the pool in read\-write mode. If this
feature is not read-only compatible, support is also required to import the pool
in read\-only mode (see "Read\-only compatibility").
.RE

.sp
.ne 2
.na
\fB\fBenabled\fR\fR
.ad
.RS 12n
An administrator has marked this feature as enabled on the pool, but the
feature's on\-disk format changes have not been made yet. The pool can still be
imported by software that does not support this feature, but changes may be made
to the on\-disk format at any time which will move the feature to the
\fBactive\fR state. Some features may support returning to the \fBenabled\fR
state after becoming \fBactive\fR. See feature\-specific documentation for
details.
.RE

.sp
.ne 2
.na
\fBdisabled\fR
.ad
.RS 12n
This feature's on\-disk format changes have not been made and will not be made
unless an administrator moves the feature to the \fBenabled\fR state. Features
cannot be disabled once they have been enabled.
.RE

.sp
.LP
The state of supported features is exposed through pool properties of the form
\fIfeature@short_name\fR.
.SS "Read\-only compatibility"
.sp
.LP
Some features may make on\-disk format changes that do not interfere with other
software's ability to read from the pool. These features are referred to as
"read\-only compatible". If all unsupported features on a pool are read\-only
compatible, the pool can be imported in read\-only mode by setting the
\fBreadonly\fR property during import (see \fBzpool\fR(1M) for details on
importing pools).
.SS "Unsupported features"
.sp
.LP
For each unsupported feature enabled on an imported pool a pool property
named \fIunsupported@feature_guid\fR will indicate why the import was allowed
despite the unsupported feature. Possible values for this property are:

.sp
.ne 2
.na
\fB\fBinactive\fR\fR
.ad
.RS 12n
The feature is in the \fBenabled\fR state and therefore the pool's on\-disk
format is still compatible with software that does not support this feature.
.RE
```

```
.sp
.ne 2
.na
\fB\fBreadonly\fR\fR
.ad
.RS 12n
The feature is read\-only compatible and the pool has been imported in
read\-only mode.
.RE

.SS "Feature dependencies"
.sp
.LP
Some features depend on other features being enabled in order to function
properly. Enabling a feature will automatically enable any features it
depends on.
.SH FEATURES
.sp
.LP
The following features are supported on this system:
.sp
.ne 2
.na
\fB\fBasync_destroy\fR\fR
.ad
.RS 4n
.TS
l l .
GUID    com.delphix:async_destroy
READ\-ONLY COMPATIBLE   yes
DEPENDENCIES    none
.TE

Destroying a file system requires traversing all of its data in order to
return its used space to the pool. Without \fBasync_destroy\fR the file system
is not fully removed until all space has been reclaimed. If the destroy
operation is interrupted by a reboot or power outage the next attempt to open
the pool will need to complete the destroy operation synchronously.

When \fBasync_destroy\fR is enabled the file system's data will be reclaimed
by a background process, allowing the destroy operation to complete without
traversing the entire file system. The background process is able to resume
interrupted destroys after the pool has been opened, eliminating the need
to finish interrupted destroys as part of the open operation. The amount
of space remaining to be reclaimed by the background process is available
through the \fBfreeing\fR property.

This feature is only \fBactive\fR while \fBfreeing\fR is non\-zero.
.RE

.sp
.ne 2
.na
\fB\fBempty_bpobj\fR\fR
.ad
.RS 4n
.TS
l l .
GUID    com.delphix:empty_bpobj
READ\-ONLY COMPATIBLE   yes
DEPENDENCIES    none
.TE

This feature increases the performance of creating and using a large
number of snapshots of a single filesystem or volume, and also reduces
```



```
the disk space required.

When there are many snapshots, each snapshot uses many Block Pointer
Objects (bpobj's) to track blocks associated with that snapshot.
However, in common use cases, most of these bpobj's are empty.  This
feature allows us to create each bpobj on-demand, thus eliminating the
empty bpobjs.

This feature is \fBactive\fR while there are any filesystems, volumes,
or snapshots which were created after enabling this feature.
.RE

.sp
.ne 2
.na
\fB\fBlz4_compress\fR\fR
.ad
.RS 4n
.TS
l l .
GUID    org.illumos:lz4_compress
READ\-ONLY COMPATIBLE   no
DEPENDENCIES    none
.TE

\fBlz4\fR is a high-performance real-time compression algorithm that
features significantly faster compression and decompression as well as a
higher compression ratio than the older \fBlzjb\fR compression.
Typically, \fBlz4\fR compression is approximately 50% faster on
compressible data and 200% faster on incompressible data than
\fBlzjb\fR. It is also approximately 80% faster on decompression, while
giving approximately 10% better compression ratio.

When the \fBlz4_compress\fR feature is set to \fBenabled\fR, the
administrator can turn on \fBlz4\fR compression on any dataset on the
pool using the \fBzfs\fR(1M) command. Please note that doing so will
immediately activate the \fBlz4_compress\fR feature on the underlying
pool (even before any data is written). Since this feature is not
read-only compatible, this operation will render the pool unimportable
on systems without support for the \fBlz4_compress\fR feature. At the
moment, this operation cannot be reversed. Booting off of
\fBlz4\fR-compressed root pools is supported.
.RE

.sp
.ne 2
.na
\fB\fBspacemap_histogram\fR\fR
.ad
.RS 4n
.TS
l l .
GUID    com.delphix:spacemap_histogram
READ\-ONLY COMPATIBLE   yes
DEPENDENCIES    none
.TE

This features allows ZFS to maintain more information about how free space
is organized within the pool. If this feature is \fBenabled\fR, ZFS will
set this feature to \fBactive\fR when a new space map object is created or
an existing space map is upgraded to the new format. Once the feature is
\fBactive\fR, it will remain in that state until the pool is destroyed.
.RE

.sp
.ne 2
```

```
.na
\fB\fBmulti_vdev_crash_dump\fR\fR
.ad
.RS 4n
.TS
l l .
GUID    com.joyent:multi_vdev_crash_dump
READ\-ONLY COMPATIBLE   no
DEPENDENCIES    none
.TE

This feature allows a dump device to be configured with a pool comprised
of multiple vdevs.  Those vdevs may be arranged in any mirrored or raidz
configuration.

When the \fBmulti_vdev_crash_dump\fR feature is set to \fBenabled\fR,
the administrator can use the \fBdumpadm\fR(1M) command to configure a
dump device on a pool comprised of multiple vdevs.
.RE

.sp
.ne 2
.na
\fB\fBextensible_dataset\fR\fR
.ad
.RS 4n
.TS
l l .
GUID    com.delphix:extensible_dataset
READ\-ONLY COMPATIBLE   no
DEPENDENCIES    none
.TE

This feature allows more flexible use of internal ZFS data structures,
and exists for other features to depend on.

This feature will be \fBactive\fR when the first dependent feature uses it,
and will be returned to the \fBenabled\fR state when all datasets that use
this feature are destroyed.

.RE

.SH "SEE ALSO"
\fBzpool\fR(1M)
```

```
**********************************************************
   15097 Tue Oct  1 14:04:24 2013
new/usr/src/uts/common/fs/zfs/bpobj.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/bpobj.h>
#include <sys/zfs_context.h>
#include <sys/refcount.h>
#include <sys/dsl_pool.h>
#include <sys/zfeature.h>
#include <sys/zap.h>

/*
 * Return an empty bpobj, preferably the empty dummy one (dp_empty_bpobj).
 */
uint64_t
bpobj_alloc_empty(objset_t *os, int blocksize, dmu_tx_t *tx)
{
        zfeature_info_t *empty_bpobj_feat =
            &spa_feature_table[SPA_FEATURE_EMPTY_BPOBJ];
        spa_t *spa = dmu_objset_spa(os);
        dsl_pool_t *dp = dmu_objset_pool(os);

        if (spa_feature_is_enabled(spa, SPA_FEATURE_EMPTY_BPOBJ)) {
                if (!spa_feature_is_active(spa, SPA_FEATURE_EMPTY_BPOBJ)) {
        if (spa_feature_is_enabled(spa, empty_bpobj_feat)) {
                if (!spa_feature_is_active(spa, empty_bpobj_feat)) {
                        ASSERT0(dp->dp_empty_bpobj);
                        dp->dp_empty_bpobj =
                            bpobj_alloc(os, SPA_MAXBLOCKSIZE, tx);
                        VERIFY(zap_add(os,
                            DMU_POOL_DIRECTORY_OBJECT,
                            DMU_POOL_EMPTY_BPOBJ, sizeof (uint64_t), 1,
                            &dp->dp_empty_bpobj, tx) == 0);
                }
                spa_feature_incr(spa, SPA_FEATURE_EMPTY_BPOBJ, tx);
                spa_feature_incr(spa, empty_bpobj_feat, tx);
```



```
                ASSERT(dp->dp_empty_bpobj != 0);
                return (dp->dp_empty_bpobj);
        } else {
                return (bpobj_alloc(os, blocksize, tx));
        }
}

void
bpobj_decr_empty(objset_t *os, dmu_tx_t *tx)
{
        zfeature_info_t *empty_bpobj_feat =
            &spa_feature_table[SPA_FEATURE_EMPTY_BPOBJ];
        dsl_pool_t *dp = dmu_objset_pool(os);

        spa_feature_decr(dmu_objset_spa(os), SPA_FEATURE_EMPTY_BPOBJ, tx);
        if (!spa_feature_is_active(dmu_objset_spa(os),
            SPA_FEATURE_EMPTY_BPOBJ)) {
        spa_feature_decr(dmu_objset_spa(os), empty_bpobj_feat, tx);
        if (!spa_feature_is_active(dmu_objset_spa(os), empty_bpobj_feat)) {
                VERIFY3U(0, ==, zap_remove(dp->dp_meta_objset,
                    DMU_POOL_DIRECTORY_OBJECT,
                    DMU_POOL_EMPTY_BPOBJ, tx));
                VERIFY3U(0, ==, dmu_object_free(os, dp->dp_empty_bpobj, tx));
                dp->dp_empty_bpobj = 0;
        }
}
_____unchanged_portion_omitted_

static int
bpobj_iterate_impl(bpobj_t *bpo, bpobj_itor_t func, void *arg, dmu_tx_t *tx,
    boolean_t free)
{
        dmu_object_info_t doi;
        int epb;
        int64_t i;
        int err = 0;
        dmu_buf_t *dbuf = NULL;

        mutex_enter(&bpo->bpo_lock);

        if (free)
                dmu_buf_will_dirty(bpo->bpo_dbuf, tx);

        for (i = bpo->bpo_phys->bpo_num_blkptrs - 1; i >= 0; i--) {
                blkptr_t *bparray;
                blkptr_t *bp;
                uint64_t offset, blkoff;

                offset = i * sizeof (blkptr_t);
                blkoff = P2PHASE(i, bpo->bpo_epb);

                if (dbuf == NULL || dbuf->db_offset > offset) {
                        if (dbuf)
                                dmu_buf_rele(dbuf, FTAG);
                        err = dmu_buf_hold(bpo->bpo_os, bpo->bpo_object, offset,
                            FTAG, &dbuf, 0);
                        if (err)
                                break;
                }

                ASSERT3U(offset, >=, dbuf->db_offset);
                ASSERT3U(offset, <, dbuf->db_offset + dbuf->db_size);

                bparray = dbuf->db_data;
                bp = &bparray[blkoff];
                err = func(arg, bp, tx);
```

```
233                 if (err)
234                         break;
235                 if (free) {
236                         bpo->bpo_phys->bpo_bytes -=
237                             bp_get_dsize_sync(dmu_objset_spa(bpo->bpo_os), bp);
238                         ASSERT3S(bpo->bpo_phys->bpo_bytes, >=, 0);
239                         if (bpo->bpo_havecomp) {
240                                 bpo->bpo_phys->bpo_comp -= BP_GET_PSIZE(bp);
241                                 bpo->bpo_phys->bpo_uncomp -= BP_GET_UCSIZE(bp);
242                         }
243                         bpo->bpo_phys->bpo_num_blkptrs--;
244                         ASSERT3S(bpo->bpo_phys->bpo_num_blkptrs, >=, 0);
245                 }
246         }
247         if (dbuf) {
248                 dmu_buf_rele(dbuf, FTAG);
249                 dbuf = NULL;
250         }
251         if (free) {
252                 i++;
253                 VERIFY3U(0, ==, dmu_free_range(bpo->bpo_os, bpo->bpo_object,
254                     i * sizeof (blkptr_t), -1ULL, tx));
255         }
256         if (err || !bpo->bpo_havesubobj || bpo->bpo_phys->bpo_subobjs == 0)
257                 goto out;

259         ASSERT(bpo->bpo_havecomp);
260         err = dmu_object_info(bpo->bpo_os, bpo->bpo_phys->bpo_subobjs, &doi);
261         if (err) {
262                 mutex_exit(&bpo->bpo_lock);
263                 return (err);
264         }
265         ASSERT3U(doi.doi_type, ==, DMU_OT_BPOBJ_SUBOBJ);
266         epb = doi.doi_data_block_size / sizeof (uint64_t);

268         for (i = bpo->bpo_phys->bpo_num_subobjs - 1; i >= 0; i--) {
269                 uint64_t *objarray;
270                 uint64_t offset, blkoff;
271                 bpobj_t sublist;
272                 uint64_t used_before, comp_before, uncomp_before;
273                 uint64_t used_after, comp_after, uncomp_after;

275                 offset = i * sizeof (uint64_t);
276                 blkoff = P2PHASE(i, epb);

278                 if (dbuf == NULL || dbuf->db_offset > offset) {
279                         if (dbuf)
280                                 dmu_buf_rele(dbuf, FTAG);
281                         err = dmu_buf_hold(bpo->bpo_os,
282                             bpo->bpo_phys->bpo_subobjs, offset, FTAG, &dbuf, 0);
283                         if (err)
284                                 break;
285                 }

287                 ASSERT3U(offset, >=, dbuf->db_offset);
288                 ASSERT3U(offset, <, dbuf->db_offset + dbuf->db_size);

290                 objarray = dbuf->db_data;
291                 err = bpobj_open(&sublist, bpo->bpo_os, objarray[blkoff]);
292                 if (err)
293                         break;
294                 if (free) {
295                         err = bpobj_space(&sublist,
296                             &used_before, &comp_before, &uncomp_before);
297                         if (err)
298                                 break;
```



```
299                 }
300                 err = bpobj_iterate_impl(&sublist, func, arg, tx, free);
301                 if (free) {
302                         VERIFY3U(0, ==, bpobj_space(&sublist,
303                             &used_after, &comp_after, &uncomp_after));
304                         bpo->bpo_phys->bpo_bytes -= used_before - used_after;
305                         ASSERT3S(bpo->bpo_phys->bpo_bytes, >=, 0);
306                         bpo->bpo_phys->bpo_comp -= comp_before - comp_after;
307                         bpo->bpo_phys->bpo_uncomp -=
308                             uncomp_before - uncomp_after;
309                 }

311                 bpobj_close(&sublist);
312                 if (err)
313                         break;
314                 if (free) {
315                         err = dmu_object_free(bpo->bpo_os,
316                             objarray[blkoff], tx);
317                         if (err)
318                                 break;
319                         bpo->bpo_phys->bpo_num_subobjs--;
320                         ASSERT3S(bpo->bpo_phys->bpo_num_subobjs, >=, 0);
321                 }
322         }
323         if (dbuf) {
324                 dmu_buf_rele(dbuf, FTAG);
325                 dbuf = NULL;
326         }
327         if (free) {
328                 VERIFY3U(0, ==, dmu_free_range(bpo->bpo_os,
329                     bpo->bpo_phys->bpo_subobjs,
330                     (i + 1) * sizeof (uint64_t), -1ULL, tx));
331         }

333 out:
334         /* If there are no entries, there should be no bytes. */
335         ASSERT(bpo->bpo_phys->bpo_num_blkptrs > 0 ||
336             (bpo->bpo_havesubobj && bpo->bpo_phys->bpo_num_subobjs > 0) ||
337             bpo->bpo_phys->bpo_bytes == 0);

339         mutex_exit(&bpo->bpo_lock);
340         return (err);
341 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
    6467 Tue Oct  1 14:04:26 2013
new/usr/src/uts/common/fs/zfs/dmu_object.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dnode.h>
#include <sys/zap.h>
#include <sys/zfeature.h>

uint64_t
dmu_object_alloc(objset_t *os, dmu_object_type_t ot, int blocksize,
    dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx)
{
        uint64_t object;
        uint64_t L2_dnode_count = DNODES_PER_BLOCK <<
            (DMU_META_DNODE(os)->dn_indblkshift - SPA_BLKPTRSHIFT);
        dnode_t *dn = NULL;
        int restarted = B_FALSE;

        mutex_enter(&os->os_obj_lock);
        for (;;) {
                object = os->os_obj_next;
                /*
                 * Each time we polish off an L2 bp worth of dnodes
                 * (2^13 objects), move to another L2 bp that's still
                 * reasonably sparse (at most 1/4 full).  Look from the
                 * beginning once, but after that keep looking from here.
                 * If we can't find one, just keep going from here.
                 */
                if (P2PHASE(object, L2_dnode_count) == 0) {
                        uint64_t offset = restarted ? object << DNODE_SHIFT : 0;
                        int error = dnode_next_offset(DMU_META_DNODE(os),
                            DNODE_FIND_HOLE,
                            &offset, 2, DNODES_PER_BLOCK >> 2, 0);
```



```
                        restarted = B_TRUE;
                        if (error == 0)
                                object = offset >> DNODE_SHIFT;
                }
                os->os_obj_next = ++object;

                /*
                 * XXX We should check for an i/o error here and return
                 * up to our caller.  Actually we should pre-read it in
                 * dmu_tx_assign(), but there is currently no mechanism
                 * to do so.
                 */
                (void) dnode_hold_impl(os, object, DNODE_MUST_BE_FREE,
                    FTAG, &dn);
                if (dn)
                        break;

                if (dmu_object_next(os, &object, B_TRUE, 0) == 0)
                        os->os_obj_next = object - 1;
        }

        dnode_allocate(dn, ot, blocksize, 0, bonustype, bonuslen, tx);
        dnode_rele(dn, FTAG);

        mutex_exit(&os->os_obj_lock);

        dmu_tx_add_new_object(tx, os, object);
        return (object);
}
_____unchanged_portion_omitted_

int
dmu_object_next(objset_t *os, uint64_t *objectp, boolean_t hole, uint64_t txg)
{
        uint64_t offset = (*objectp + 1) << DNODE_SHIFT;
        int error;

        error = dnode_next_offset(DMU_META_DNODE(os),
            (hole ? DNODE_FIND_HOLE : 0), &offset, 0, DNODES_PER_BLOCK, txg);

        *objectp = offset >> DNODE_SHIFT;

        return (error);
}

/*
 * Turn this object from old_type into DMU_OTN_ZAP_METADATA, and bump the
 * refcount on SPA_FEATURE_EXTENSIBLE_DATASET.
 *
 * Only for use from syncing context, on MOS objects.
 */
void
dmu_object_zapify(objset_t *mos, uint64_t object, dmu_object_type_t old_type,
    dmu_tx_t *tx)
{
        dnode_t *dn;

        ASSERT(dmu_tx_is_syncing(tx));

        VERIFY0(dnode_hold(mos, object, FTAG, &dn));
        if (dn->dn_type == DMU_OTN_ZAP_METADATA) {
                dnode_rele(dn, FTAG);
                return;
        }
        ASSERT3U(dn->dn_type, ==, old_type);
        ASSERT0(dn->dn_maxblkid);
```

```
222         dn->dn_next_type[tx->tx_txg & TXG_MASK] = dn->dn_type =
223             DMU_OTN_ZAP_METADATA;
224         dnode_setdirty(dn, tx);
225         dnode_rele(dn, FTAG);

227         mzap_create_impl(mos, object, 0, 0, tx);

229         spa_feature_incr(dmu_objset_spa(mos),
230             SPA_FEATURE_EXTENSIBLE_DATASET, tx);
231 }

233 void
234 dmu_object_free_zapified(objset_t *mos, uint64_t object, dmu_tx_t *tx)
235 {
236         dnode_t *dn;
237         dmu_object_type_t t;

239         ASSERT(dmu_tx_is_syncing(tx));

241         VERIFY0(dnode_hold(mos, object, FTAG, &dn));
242         t = dn->dn_type;
243         dnode_rele(dn, FTAG);

245         if (t == DMU_OTN_ZAP_METADATA) {
246                 spa_feature_decr(dmu_objset_spa(mos),
247                     SPA_FEATURE_EXTENSIBLE_DATASET, tx);
248         }
249         VERIFY0(dmu_object_free(mos, object, tx));
250 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   16666 Tue Oct  1 14:04:28 2013
new/usr/src/uts/common/fs/zfs/dmu_traverse.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 576 /*
 577  * NB: pool must not be changing on-disk (eg, from zdb or sync context).
 578  */
 579 int
 580 traverse_pool(spa_t *spa, uint64_t txg_start, int flags,
 581     blkptr_cb_t func, void *arg)
 582 {
 583         int err, lasterr = 0;
 584         uint64_t obj;
 585         dsl_pool_t *dp = spa_get_dsl(spa);
 586         objset_t *mos = dp->dp_meta_objset;
 587         boolean_t hard = (flags & TRAVERSE_HARD);

 589         /* visit the MOS */
 590         err = traverse_impl(spa, NULL, 0, spa_get_rootblkptr(spa),
 591             txg_start, NULL, flags, func, arg);
 592         if (err != 0)
 593                 return (err);

 595         /* visit each dataset */
 596         for (obj = 1; err == 0 || (err != ESRCH && hard);
 597             err = dmu_object_next(mos, &obj, FALSE, txg_start)) {
 598                 dmu_object_info_t doi;

 600                 err = dmu_object_info(mos, obj, &doi);
 601                 if (err != 0) {
 602                         if (!hard)
 603                                 return (err);
 604                         lasterr = err;
 605                         continue;
 606                 }

 608                 if (doi.doi_bonus_type == DMU_OT_DSL_DATASET) {
 608                 if (doi.doi_type == DMU_OT_DSL_DATASET) {
 609                         dsl_dataset_t *ds;
 610                         uint64_t txg = txg_start;

 612                         dsl_pool_config_enter(dp, FTAG);
 613                         err = dsl_dataset_hold_obj(dp, obj, FTAG, &ds);
 614                         dsl_pool_config_exit(dp, FTAG);
 615                         if (err != 0) {
 616                                 if (!hard)
 617                                         return (err);
 618                                 lasterr = err;
 619                                 continue;
 620                         }
 621                         if (ds->ds_phys->ds_prev_snap_txg > txg)
 622                                 txg = ds->ds_phys->ds_prev_snap_txg;
 623                         err = traverse_dataset(ds, txg, flags, func, arg);
 624                         dsl_dataset_rele(ds, FTAG);
 625                         if (err != 0) {
 626                                 if (!hard)
 627                                         return (err);
 628                                 lasterr = err;
 629                         }
```



```
 630                 }
 631         }
 632         if (err == ESRCH)
 633                 err = 0;
 634         return (err != 0 ? err : lasterr);
 635 }
______unchanged_portion_omitted_
```

```
**********************************************************
   19442 Tue Oct  1 14:04:30 2013
new/usr/src/uts/common/fs/zfs/dnode_sync.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */

/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/dbuf.h>
#include <sys/dnode.h>
#include <sys/dmu.h>
#include <sys/dmu_tx.h>
#include <sys/dmu_objset.h>
#include <sys/dsl_dataset.h>
#include <sys/spa.h>

static void
dnode_increase_indirection(dnode_t *dn, dmu_tx_t *tx)
{
        dmu_buf_impl_t *db;
        int txgoff = tx->tx_txg & TXG_MASK;
        int nblkptr = dn->dn_phys->dn_nblkptr;
        int old_toplvl = dn->dn_phys->dn_nlevels - 1;
        int new_level = dn->dn_next_nlevels[txgoff];
        int i;

        rw_enter(&dn->dn_struct_rwlock, RW_WRITER);

        /* this dnode can't be paged out because it's dirty */
        ASSERT(dn->dn_phys->dn_type != DMU_OT_NONE);
        ASSERT(RW_WRITE_HELD(&dn->dn_struct_rwlock));
        ASSERT(new_level > 1 && dn->dn_phys->dn_nlevels > 0);

        db = dbuf_hold_level(dn, dn->dn_phys->dn_nlevels, 0, FTAG);
        ASSERT(db != NULL);

        dn->dn_phys->dn_nlevels = new_level;
```



```
        dprintf("os=%p obj=%llu, increase to %d\n", dn->dn_objset,
            dn->dn_object, dn->dn_phys->dn_nlevels);

        /* check for existing blkptrs in the dnode */
        for (i = 0; i < nblkptr; i++)
                if (!BP_IS_HOLE(&dn->dn_phys->dn_blkptr[i]))
                        break;
        if (i != nblkptr) {
                /* transfer dnode's block pointers to new indirect block */
                (void) dbuf_read(db, NULL, DB_RF_MUST_SUCCEED|DB_RF_HAVESTRUCT);
                ASSERT(db->db.db_data);
                ASSERT(arc_released(db->db_buf));
                ASSERT3U(sizeof (blkptr_t) * nblkptr, <=, db->db.db_size);
                bcopy(dn->dn_phys->dn_blkptr, db->db.db_data,
                    sizeof (blkptr_t) * nblkptr);
                arc_buf_freeze(db->db_buf);
        }

        /* set dbuf's parent pointers to new indirect buf */
        for (i = 0; i < nblkptr; i++) {
                dmu_buf_impl_t *child = dbuf_find(dn, old_toplvl, i);

                if (child == NULL)
                        continue;
#ifdef  DEBUG
                DB_DNODE_ENTER(child);
                ASSERT3P(DB_DNODE(child), ==, dn);
                DB_DNODE_EXIT(child);
#endif  /* DEBUG */
                if (child->db_parent && child->db_parent != dn->dn_dbuf) {
                        ASSERT(child->db_parent->db_level == db->db_level);
                        ASSERT(child->db_blkptr !=
                            &dn->dn_phys->dn_blkptr[child->db_blkid]);
                        mutex_exit(&child->db_mtx);
                        continue;
                }
                ASSERT(child->db_parent == NULL ||
                    child->db_parent == dn->dn_dbuf);

                child->db_parent = db;
                dbuf_add_ref(db, child);
                if (db->db.db_data)
                        child->db_blkptr = (blkptr_t *)db->db.db_data + i;
                else
                        child->db_blkptr = NULL;
                dprintf_dbuf_bp(child, child->db_blkptr,
                    "changed db_blkptr to new indirect %s", "");

                mutex_exit(&child->db_mtx);
        }

        bzero(dn->dn_phys->dn_blkptr, sizeof (blkptr_t) * nblkptr);

        dbuf_rele(db, FTAG);

        rw_exit(&dn->dn_struct_rwlock);
}
_____unchanged_portion_omitted_

/*
 * Write out the dnode's dirty buffers.
 */
void
dnode_sync(dnode_t *dn, dmu_tx_t *tx)
{
        free_range_t *rp;
```

```
530         dnode_phys_t *dnp = dn->dn_phys;
531         int txgoff = tx->tx_txg & TXG_MASK;
532         list_t *list = &dn->dn_dirty_records[txgoff];
533         static const dnode_phys_t zerodn = { 0 };
534         boolean_t kill_spill = B_FALSE;

536         ASSERT(dmu_tx_is_syncing(tx));
537         ASSERT(dnp->dn_type != DMU_OT_NONE || dn->dn_allocated_txg);
538         ASSERT(dnp->dn_type != DMU_OT_NONE ||
539             bcmp(dnp, &zerodn, DNODE_SIZE) == 0);
540         DNODE_VERIFY(dn);

542         ASSERT(dn->dn_dbuf == NULL || arc_released(dn->dn_dbuf->db_buf));

544         if (dmu_objset_userused_enabled(dn->dn_objset) &&
545             !DMU_OBJECT_IS_SPECIAL(dn->dn_object)) {
546                 mutex_enter(&dn->dn_mtx);
547                 dn->dn_oldused = DN_USED_BYTES(dn->dn_phys);
548                 dn->dn_oldflags = dn->dn_phys->dn_flags;
549                 dn->dn_phys->dn_flags |= DNODE_FLAG_USERUSED_ACCOUNTED;
550                 mutex_exit(&dn->dn_mtx);
551                 dmu_objset_userquota_get_ids(dn, B_FALSE, tx);
552         } else {
553                 /* Once we account for it, we should always account for it. */
554                 ASSERT(!(dn->dn_phys->dn_flags &
555                     DNODE_FLAG_USERUSED_ACCOUNTED));
556         }

558         mutex_enter(&dn->dn_mtx);
559         if (dn->dn_allocated_txg == tx->tx_txg) {
560                 /* The dnode is newly allocated or reallocated */
561                 if (dnp->dn_type == DMU_OT_NONE) {
562                         /* this is a first alloc, not a realloc */
563                         dnp->dn_nlevels = 1;
564                         dnp->dn_nblkptr = dn->dn_nblkptr;
565                 }

567                 dnp->dn_type = dn->dn_type;
568                 dnp->dn_bonustype = dn->dn_bonustype;
569                 dnp->dn_bonuslen = dn->dn_bonuslen;
570         }

572         ASSERT(dnp->dn_nlevels > 1 ||
573             BP_IS_HOLE(&dnp->dn_blkptr[0]) ||
574             BP_GET_LSIZE(&dnp->dn_blkptr[0]) ==
575             dnp->dn_datablkszsec << SPA_MINBLOCKSHIFT);

577         if (dn->dn_next_type[txgoff] != 0) {
578                 dnp->dn_type = dn->dn_type;
579                 dn->dn_next_type[txgoff] = 0;
580         }

582         if (dn->dn_next_blksz[txgoff] != 0) {
577         if (dn->dn_next_blksz[txgoff]) {
583                 ASSERT(P2PHASE(dn->dn_next_blksz[txgoff],
584                     SPA_MINBLOCKSIZE) == 0);
585                 ASSERT(BP_IS_HOLE(&dnp->dn_blkptr[0]) ||
586                     dn->dn_maxblkid == 0 || list_head(list) != NULL ||
587                     avl_last(&dn->dn_ranges[txgoff]) ||
588                     dn->dn_next_blksz[txgoff] >> SPA_MINBLOCKSHIFT ==
589                     dnp->dn_datablkszsec);
590                 dnp->dn_datablkszsec =
591                     dn->dn_next_blksz[txgoff] >> SPA_MINBLOCKSHIFT;
592                 dn->dn_next_blksz[txgoff] = 0;
593         }
```



```
595         if (dn->dn_next_bonuslen[txgoff] != 0) {
590         if (dn->dn_next_bonuslen[txgoff]) {
596                 if (dn->dn_next_bonuslen[txgoff] == DN_ZERO_BONUSLEN)
597                         dnp->dn_bonuslen = 0;
598                 else
599                         dnp->dn_bonuslen = dn->dn_next_bonuslen[txgoff];
600                 ASSERT(dnp->dn_bonuslen <= DN_MAX_BONUSLEN);
601                 dn->dn_next_bonuslen[txgoff] = 0;
602         }

604         if (dn->dn_next_bonustype[txgoff] != 0) {
599         if (dn->dn_next_bonustype[txgoff]) {
605                 ASSERT(DMU_OT_IS_VALID(dn->dn_next_bonustype[txgoff]));
606                 dnp->dn_bonustype = dn->dn_next_bonustype[txgoff];
607                 dn->dn_next_bonustype[txgoff] = 0;
608         }

610         /*
611          * We will either remove a spill block when a file is being removed
612          * or we have been asked to remove it.
613          */
614         if (dn->dn_rm_spillblk[txgoff] ||
615             ((dnp->dn_flags & DNODE_FLAG_SPILL_BLKPTR) &&
616             dn->dn_free_txg > 0 && dn->dn_free_txg <= tx->tx_txg)) {
617                 if ((dnp->dn_flags & DNODE_FLAG_SPILL_BLKPTR))
618                         kill_spill = B_TRUE;
619                 dn->dn_rm_spillblk[txgoff] = 0;
620         }

622         if (dn->dn_next_indblkshift[txgoff] != 0) {
617         if (dn->dn_next_indblkshift[txgoff]) {
623                 ASSERT(dnp->dn_nlevels == 1);
624                 dnp->dn_indblkshift = dn->dn_next_indblkshift[txgoff];
625                 dn->dn_next_indblkshift[txgoff] = 0;
626         }

628         /*
629          * Just take the live (open-context) values for checksum and compress.
630          * Strictly speaking it's a future leak, but nothing bad happens if we
631          * start using the new checksum or compress algorithm a little early.
632          */
633         dnp->dn_checksum = dn->dn_checksum;
634         dnp->dn_compress = dn->dn_compress;

636         mutex_exit(&dn->dn_mtx);

638         if (kill_spill) {
639                 (void) free_blocks(dn, &dn->dn_phys->dn_spill, 1, tx);
640                 mutex_enter(&dn->dn_mtx);
641                 dnp->dn_flags &= ~DNODE_FLAG_SPILL_BLKPTR;
642                 mutex_exit(&dn->dn_mtx);
643         }

645         /* process all the "freed" ranges in the file */
646         while (rp = avl_last(&dn->dn_ranges[txgoff])) {
647                 dnode_sync_free_range(dn, rp->fr_blkid, rp->fr_nblks, tx);
648                 /* grab the mutex so we don't race with dnode_block_freed() */
649                 mutex_enter(&dn->dn_mtx);
650                 avl_remove(&dn->dn_ranges[txgoff], rp);
651                 mutex_exit(&dn->dn_mtx);
652                 kmem_free(rp, sizeof (free_range_t));
653         }

655         if (dn->dn_free_txg > 0 && dn->dn_free_txg <= tx->tx_txg) {
656                 dnode_sync_free(dn, tx);
657                 return;
```

```
658         }

660         if (dn->dn_next_nblkptr[txgoff]) {
661                 /* this should only happen on a realloc */
662                 ASSERT(dn->dn_allocated_txg == tx->tx_txg);
663                 if (dn->dn_next_nblkptr[txgoff] > dnp->dn_nblkptr) {
664                         /* zero the new blkptrs we are gaining */
665                         bzero(dnp->dn_blkptr + dnp->dn_nblkptr,
666                             sizeof (blkptr_t) *
667                             (dn->dn_next_nblkptr[txgoff] - dnp->dn_nblkptr));
668 #ifdef ZFS_DEBUG
669                 } else {
670                         int i;
671                         ASSERT(dn->dn_next_nblkptr[txgoff] < dnp->dn_nblkptr);
672                         /* the blkptrs we are losing better be unallocated */
673                         for (i = dn->dn_next_nblkptr[txgoff];
674                             i < dnp->dn_nblkptr; i++)
675                                 ASSERT(BP_IS_HOLE(&dnp->dn_blkptr[i]));
676 #endif
677                 }
678                 mutex_enter(&dn->dn_mtx);
679                 dnp->dn_nblkptr = dn->dn_next_nblkptr[txgoff];
680                 dn->dn_next_nblkptr[txgoff] = 0;
681                 mutex_exit(&dn->dn_mtx);
682         }

684         if (dn->dn_next_nlevels[txgoff]) {
685                 dnode_increase_indirection(dn, tx);
686                 dn->dn_next_nlevels[txgoff] = 0;
687         }

689         dbuf_sync_list(list, tx);

691         if (!DMU_OBJECT_IS_SPECIAL(dn->dn_object)) {
692                 ASSERT3P(list_head(list), ==, NULL);
693                 dnode_rele(dn, (void *)(uintptr_t)tx->tx_txg);
694         }

696         /*
697          * Although we have dropped our reference to the dnode, it
698          * can't be evicted until its written, and we haven't yet
699          * initiated the IO for the dnode's dbuf.
700          */
701 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
   82608 Tue Oct  1 14:04:31 2013
new/usr/src/uts/common/fs/zfs/dsl_dataset.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 337 int
 338 dsl_dataset_hold_obj(dsl_pool_t *dp, uint64_t dsobj, void *tag,
 339     dsl_dataset_t **dsp)
 340 {
 341         objset_t *mos = dp->dp_meta_objset;
 342         dmu_buf_t *dbuf;
 343         dsl_dataset_t *ds;
 344         int err;
 345         dmu_object_info_t doi;

 347         ASSERT(dsl_pool_config_held(dp));

 349         err = dmu_bonus_hold(mos, dsobj, tag, &dbuf);
 350         if (err != 0)
 351                 return (err);

 353         /* Make sure dsobj has the correct object type. */
 354         dmu_object_info_from_db(dbuf, &doi);
 355         if (doi.doi_bonus_type != DMU_OT_DSL_DATASET) {
 355         if (doi.doi_type != DMU_OT_DSL_DATASET) {
 356                 dmu_buf_rele(dbuf, tag);
 357                 return (SET_ERROR(EINVAL));
 358         }

 360         ds = dmu_buf_get_user(dbuf);
 361         if (ds == NULL) {
 362                 dsl_dataset_t *winner = NULL;

 364                 ds = kmem_zalloc(sizeof (dsl_dataset_t), KM_SLEEP);
 365                 ds->ds_dbuf = dbuf;
 366                 ds->ds_object = dsobj;
 367                 ds->ds_phys = dbuf->db_data;

 369                 mutex_init(&ds->ds_lock, NULL, MUTEX_DEFAULT, NULL);
 370                 mutex_init(&ds->ds_opening_lock, NULL, MUTEX_DEFAULT, NULL);
 371                 mutex_init(&ds->ds_sendstream_lock, NULL, MUTEX_DEFAULT, NULL);
 372                 refcount_create(&ds->ds_longholds);

 374                 bplist_create(&ds->ds_pending_deadlist);
 375                 dsl_deadlist_open(&ds->ds_deadlist,
 376                     mos, ds->ds_phys->ds_deadlist_obj);

 378                 list_create(&ds->ds_sendstreams, sizeof (dmu_sendarg_t),
 379                     offsetof(dmu_sendarg_t, dsa_link));

 381                 if (err == 0) {
 382                         err = dsl_dir_hold_obj(dp,
 383                             ds->ds_phys->ds_dir_obj, NULL, ds, &ds->ds_dir);
 384                 }
 385                 if (err != 0) {
 386                         mutex_destroy(&ds->ds_lock);
 387                         mutex_destroy(&ds->ds_opening_lock);
 388                         refcount_destroy(&ds->ds_longholds);
 389                         bplist_destroy(&ds->ds_pending_deadlist);
 390                         dsl_deadlist_close(&ds->ds_deadlist);
```



```
 391                         kmem_free(ds, sizeof (dsl_dataset_t));
 392                         dmu_buf_rele(dbuf, tag);
 393                         return (err);
 394                 }

 396                 if (!dsl_dataset_is_snapshot(ds)) {
 397                         ds->ds_snapname[0] = '\0';
 398                         if (ds->ds_phys->ds_prev_snap_obj != 0) {
 399                                 err = dsl_dataset_hold_obj(dp,
 400                                     ds->ds_phys->ds_prev_snap_obj,
 401                                     ds, &ds->ds_prev);
 402                         }
 403                 } else {
 404                         if (zfs_flags & ZFS_DEBUG_SNAPNAMES)
 405                                 err = dsl_dataset_get_snapname(ds);
 406                         if (err == 0 && ds->ds_phys->ds_userrefs_obj != 0) {
 407                                 err = zap_count(
 408                                     ds->ds_dir->dd_pool->dp_meta_objset,
 409                                     ds->ds_phys->ds_userrefs_obj,
 410                                     &ds->ds_userrefs);
 411                         }
 412                 }

 414                 if (err == 0 && !dsl_dataset_is_snapshot(ds)) {
 415                         err = dsl_prop_get_int_ds(ds,
 416                             zfs_prop_to_name(ZFS_PROP_REFRESERVATION),
 417                             &ds->ds_reserved);
 418                         if (err == 0) {
 419                                 err = dsl_prop_get_int_ds(ds,
 420                                     zfs_prop_to_name(ZFS_PROP_REFQUOTA),
 421                                     &ds->ds_quota);
 422                         }
 423                 } else {
 424                         ds->ds_reserved = ds->ds_quota = 0;
 425                 }

 427                 if (err != 0 || (winner = dmu_buf_set_user_ie(dbuf, ds,
 428                     &ds->ds_phys, dsl_dataset_evict)) != NULL) {
 429                         bplist_destroy(&ds->ds_pending_deadlist);
 430                         dsl_deadlist_close(&ds->ds_deadlist);
 431                         if (ds->ds_prev)
 432                                 dsl_dataset_rele(ds->ds_prev, ds);
 433                         dsl_dir_rele(ds->ds_dir, ds);
 434                         mutex_destroy(&ds->ds_lock);
 435                         mutex_destroy(&ds->ds_opening_lock);
 436                         refcount_destroy(&ds->ds_longholds);
 437                         kmem_free(ds, sizeof (dsl_dataset_t));
 438                         if (err != 0) {
 439                                 dmu_buf_rele(dbuf, tag);
 440                                 return (err);
 441                         }
 442                         ds = winner;
 443                 } else {
 444                         ds->ds_fsid_guid =
 445                             unique_insert(ds->ds_phys->ds_fsid_guid);
 446                 }
 447         }
 448         ASSERT3P(ds->ds_dbuf, ==, dbuf);
 449         ASSERT3P(ds->ds_phys, ==, dbuf->db_data);
 450         ASSERT(ds->ds_phys->ds_prev_snap_obj != 0 ||
 451             spa_version(dp->dp_spa) < SPA_VERSION_ORIGIN ||
 452             dp->dp_origin_snap == NULL || ds == dp->dp_origin_snap);
 453         *dsp = ds;
 454         return (0);
 455 }
______unchanged_portion_omitted_
```

```
2939 /*
2940  * Return TRUE if 'earlier' is an earlier snapshot in 'later's timeline.
2941  * For example, they could both be snapshots of the same filesystem, and
2942  * 'earlier' is before 'later'.  Or 'earlier' could be the origin of
2943  * 'later's filesystem.  Or 'earlier' could be an older snapshot in the origin's
2944  * filesystem.  Or 'earlier' could be the origin's origin.
2945  */
2946 boolean_t
2947 dsl_dataset_is_before(dsl_dataset_t *later, dsl_dataset_t *earlier)
2948 {
2949         dsl_pool_t *dp = later->ds_dir->dd_pool;
2950         int error;
2951         boolean_t ret;

2953         ASSERT(dsl_pool_config_held(dp));

2955         if (earlier->ds_phys->ds_creation_txg >=
2956             later->ds_phys->ds_creation_txg)
2957                 return (B_FALSE);

2959         if (later->ds_dir == earlier->ds_dir)
2960                 return (B_TRUE);
2961         if (!dsl_dir_is_clone(later->ds_dir))
2962                 return (B_FALSE);

2964         if (later->ds_dir->dd_phys->dd_origin_obj == earlier->ds_object)
2965                 return (B_TRUE);
2966         dsl_dataset_t *origin;
2967         error = dsl_dataset_hold_obj(dp,
2968             later->ds_dir->dd_phys->dd_origin_obj, FTAG, &origin);
2969         if (error != 0)
2970                 return (B_FALSE);
2971         ret = dsl_dataset_is_before(origin, earlier);
2972         dsl_dataset_rele(origin, FTAG);
2973         return (ret);
2974 }


2977 void
2978 dsl_dataset_zapify(dsl_dataset_t *ds, dmu_tx_t *tx)
2979 {
2980         objset_t *mos = ds->ds_dir->dd_pool->dp_meta_objset;
2981         dmu_object_zapify(mos, ds->ds_object, DMU_OT_DSL_DATASET, tx);
2982 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   25661 Tue Oct  1 14:04:33 2013
new/usr/src/uts/common/fs/zfs/dsl_destroy.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/dsl_userhold.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_synctask.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_pool.h>
#include <sys/dsl_dir.h>
#include <sys/dmu_traverse.h>
#include <sys/dsl_scan.h>
#include <sys/dmu_objset.h>
#include <sys/zap.h>
#include <sys/zfeature.h>
#include <sys/zfs_ioctl.h>
#include <sys/dsl_deleg.h>
#include <sys/dmu_impl.h>

typedef struct dmu_snapshots_destroy_arg {
        nvlist_t *dsda_snaps;
        nvlist_t *dsda_successful_snaps;
        boolean_t dsda_defer;
        nvlist_t *dsda_errlist;
} dmu_snapshots_destroy_arg_t;
```

*unchanged_portion_omitted_*

```
void
dsl_destroy_snapshot_sync_impl(dsl_dataset_t *ds, boolean_t defer, dmu_tx_t *tx)
{
        int err;
        int after_branch_point = FALSE;
        dsl_pool_t *dp = ds->ds_dir->dd_pool;
        objset_t *mos = dp->dp_meta_objset;
```



```
        dsl_dataset_t *ds_prev = NULL;
        uint64_t obj;

        ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));
        ASSERT3U(ds->ds_phys->ds_bp.blk_birth, <=, tx->tx_txg);
        ASSERT(refcount_is_zero(&ds->ds_longholds));

        if (defer &&
            (ds->ds_userrefs > 0 || ds->ds_phys->ds_num_children > 1)) {
                ASSERT(spa_version(dp->dp_spa) >= SPA_VERSION_USERREFS);
                dmu_buf_will_dirty(ds->ds_dbuf, tx);
                ds->ds_phys->ds_flags |= DS_FLAG_DEFER_DESTROY;
                spa_history_log_internal_ds(ds, "defer_destroy", tx, "");
                return;
        }

        ASSERT3U(ds->ds_phys->ds_num_children, <=, 1);

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

        dsl_scan_ds_destroyed(ds, tx);

        obj = ds->ds_object;

        if (ds->ds_phys->ds_prev_snap_obj != 0) {
                ASSERT3P(ds->ds_prev, ==, NULL);
                VERIFY0(dsl_dataset_hold_obj(dp,
                    ds->ds_phys->ds_prev_snap_obj, FTAG, &ds_prev));
                after_branch_point =
                    (ds_prev->ds_phys->ds_next_snap_obj != obj);

                dmu_buf_will_dirty(ds_prev->ds_dbuf, tx);
                if (after_branch_point &&
                    ds_prev->ds_phys->ds_next_clones_obj != 0) {
                        dsl_dataset_remove_from_next_clones(ds_prev, obj, tx);
                        if (ds->ds_phys->ds_next_snap_obj != 0) {
                                VERIFY0(zap_add_int(mos,
                                    ds_prev->ds_phys->ds_next_clones_obj,
                                    ds->ds_phys->ds_next_snap_obj, tx));
                        }
                }
                if (!after_branch_point) {
                        ds_prev->ds_phys->ds_next_snap_obj =
                            ds->ds_phys->ds_next_snap_obj;
                }
        }

        dsl_dataset_t *ds_next;
        uint64_t old_unique;
        uint64_t used = 0, comp = 0, uncomp = 0;

        VERIFY0(dsl_dataset_hold_obj(dp,
            ds->ds_phys->ds_next_snap_obj, FTAG, &ds_next));
        ASSERT3U(ds_next->ds_phys->ds_prev_snap_obj, ==, obj);

        old_unique = ds_next->ds_phys->ds_unique_bytes;

        dmu_buf_will_dirty(ds_next->ds_dbuf, tx);
        ds_next->ds_phys->ds_prev_snap_obj =
            ds->ds_phys->ds_prev_snap_obj;
        ds_next->ds_phys->ds_prev_snap_txg =
            ds->ds_phys->ds_prev_snap_txg;
        ASSERT3U(ds->ds_phys->ds_prev_snap_txg, ==,
            ds_prev ? ds_prev->ds_phys->ds_creation_txg : 0);
```

```
305         if (ds_next->ds_deadlist.dl_oldfmt) {
306                 process_old_deadlist(ds, ds_prev, ds_next,
307                     after_branch_point, tx);
308         } else {
309                 /* Adjust prev's unique space. */
310                 if (ds_prev && !after_branch_point) {
311                         dsl_deadlist_space_range(&ds_next->ds_deadlist,
312                             ds_prev->ds_phys->ds_prev_snap_txg,
313                             ds->ds_phys->ds_prev_snap_txg,
314                             &used, &comp, &uncomp);
315                         ds_prev->ds_phys->ds_unique_bytes += used;
316                 }
317
318                 /* Adjust snapused. */
319                 dsl_deadlist_space_range(&ds_next->ds_deadlist,
320                     ds->ds_phys->ds_prev_snap_txg, UINT64_MAX,
321                     &used, &comp, &uncomp);
322                 dsl_dir_diduse_space(ds->ds_dir, DD_USED_SNAP,
323                     -used, -comp, -uncomp, tx);
324
325                 /* Move blocks to be freed to pool's free list. */
326                 dsl_deadlist_move_bpobj(&ds_next->ds_deadlist,
327                     &dp->dp_free_bpobj, ds->ds_phys->ds_prev_snap_txg,
328                     tx);
329                 dsl_dir_diduse_space(tx->tx_pool->dp_free_dir,
330                     DD_USED_HEAD, used, comp, uncomp, tx);
331
332                 /* Merge our deadlist into next's and free it. */
333                 dsl_deadlist_merge(&ds_next->ds_deadlist,
334                     ds->ds_phys->ds_deadlist_obj, tx);
335         }
336         dsl_deadlist_close(&ds->ds_deadlist);
337         dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
338         dmu_buf_will_dirty(ds->ds_dbuf, tx);
339         ds->ds_phys->ds_deadlist_obj = 0;
340
341         /* Collapse range in clone heads */
342         dsl_dataset_remove_clones_key(ds,
343             ds->ds_phys->ds_creation_txg, tx);
344
345         if (dsl_dataset_is_snapshot(ds_next)) {
346                 dsl_dataset_t *ds_nextnext;
347
348                 /*
349                  * Update next's unique to include blocks which
350                  * were previously shared by only this snapshot
351                  * and it.  Those blocks will be born after the
352                  * prev snap and before this snap, and will have
353                  * died after the next snap and before the one
354                  * after that (ie. be on the snap after next's
355                  * deadlist).
356                  */
357                 VERIFY0(dsl_dataset_hold_obj(dp,
358                     ds_next->ds_phys->ds_next_snap_obj, FTAG, &ds_nextnext));
359                 dsl_deadlist_space_range(&ds_nextnext->ds_deadlist,
360                     ds->ds_phys->ds_prev_snap_txg,
361                     ds->ds_phys->ds_creation_txg,
362                     &used, &comp, &uncomp);
363                 ds_next->ds_phys->ds_unique_bytes += used;
364                 dsl_dataset_rele(ds_nextnext, FTAG);
365                 ASSERT3P(ds_next->ds_prev, ==, NULL);
366
367                 /* Collapse range in this head. */
368                 dsl_dataset_t *hds;
369                 VERIFY0(dsl_dataset_hold_obj(dp,
370                     ds->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &hds));
```



```
371                 dsl_deadlist_remove_key(&hds->ds_deadlist,
372                     ds->ds_phys->ds_creation_txg, tx);
373                 dsl_dataset_rele(hds, FTAG);
374
375         } else {
376                 ASSERT3P(ds_next->ds_prev, ==, ds);
377                 dsl_dataset_rele(ds_next->ds_prev, ds_next);
378                 ds_next->ds_prev = NULL;
379                 if (ds_prev) {
380                         VERIFY0(dsl_dataset_hold_obj(dp,
381                             ds->ds_phys->ds_prev_snap_obj,
382                             ds_next, &ds_next->ds_prev));
383                 }
384
385                 dsl_dataset_recalc_head_uniq(ds_next);
386
387                 /*
388                  * Reduce the amount of our unconsumed refreservation
389                  * being charged to our parent by the amount of
390                  * new unique data we have gained.
391                  */
392                 if (old_unique < ds_next->ds_reserved) {
393                         int64_t mrsdelta;
394                         uint64_t new_unique =
395                             ds_next->ds_phys->ds_unique_bytes;
396
397                         ASSERT(old_unique <= new_unique);
398                         mrsdelta = MIN(new_unique - old_unique,
399                             ds_next->ds_reserved - old_unique);
400                         dsl_dir_diduse_space(ds->ds_dir,
401                             DD_USED_REFRSRV, -mrsdelta, 0, 0, tx);
402                 }
403         }
404         dsl_dataset_rele(ds_next, FTAG);
405
406         /*
407          * This must be done after the dsl_traverse(), because it will
408          * re-open the objset.
409          */
410         if (ds->ds_objset) {
411                 dmu_objset_evict(ds->ds_objset);
412                 ds->ds_objset = NULL;
413         }
414
415         /* remove from snapshot namespace */
416         dsl_dataset_t *ds_head;
417         ASSERT(ds->ds_phys->ds_snapnames_zapobj == 0);
418         VERIFY0(dsl_dataset_hold_obj(dp,
419             ds->ds_dir->dd_phys->dd_head_dataset_obj, FTAG, &ds_head));
420         VERIFY0(dsl_dataset_get_snapname(ds));
421 #ifdef ZFS_DEBUG
422         {
423                 uint64_t val;
424
425                 err = dsl_dataset_snap_lookup(ds_head,
426                     ds->ds_snapname, &val);
427                 ASSERT0(err);
428                 ASSERT3U(val, ==, obj);
429         }
430 #endif
431         VERIFY0(dsl_dataset_snap_remove(ds_head, ds->ds_snapname, tx));
432         dsl_dataset_rele(ds_head, FTAG);
433
434         if (ds_prev != NULL)
435                 dsl_dataset_rele(ds_prev, FTAG);
```

```
437         spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);

439         if (ds->ds_phys->ds_next_clones_obj != 0) {
440                 uint64_t count;
441                 ASSERT0(zap_count(mos,
442                     ds->ds_phys->ds_next_clones_obj, &count) && count == 0);
443                 VERIFY0(dmu_object_free(mos,
444                     ds->ds_phys->ds_next_clones_obj, tx));
445         }
446         if (ds->ds_phys->ds_props_obj != 0)
447                 VERIFY0(zap_destroy(mos, ds->ds_phys->ds_props_obj, tx));
448         if (ds->ds_phys->ds_userrefs_obj != 0)
449                 VERIFY0(zap_destroy(mos, ds->ds_phys->ds_userrefs_obj, tx));
450         dsl_dir_rele(ds->ds_dir, ds);
451         ds->ds_dir = NULL;
452         dmu_object_free_zapified(mos, obj, tx);
451         VERIFY0(dmu_object_free(mos, obj, tx));
453 }
_____unchanged_portion_omitted_

643 static void
644 dsl_dir_destroy_sync(uint64_t ddobj, dmu_tx_t *tx)
645 {
646         dsl_dir_t *dd;
647         dsl_pool_t *dp = dmu_tx_pool(tx);
648         objset_t *mos = dp->dp_meta_objset;
649         dd_used_t t;

651         ASSERT(RRW_WRITE_HELD(&dmu_tx_pool(tx)->dp_config_rwlock));

653         VERIFY0(dsl_dir_hold_obj(dp, ddobj, NULL, FTAG, &dd));

655         ASSERT0(dd->dd_phys->dd_head_dataset_obj);

657         /*
658          * Remove our reservation. The impl() routine avoids setting the
659          * actual property, which would require the (already destroyed) ds.
660          */
661         dsl_dir_set_reservation_sync_impl(dd, 0, tx);

663         ASSERT0(dd->dd_phys->dd_used_bytes);
664         ASSERT0(dd->dd_phys->dd_reserved);
665         for (t = 0; t < DD_USED_NUM; t++)
666                 ASSERT0(dd->dd_phys->dd_used_breakdown[t]);

668         VERIFY0(zap_destroy(mos, dd->dd_phys->dd_child_dir_zapobj, tx));
669         VERIFY0(zap_destroy(mos, dd->dd_phys->dd_props_zapobj, tx));
670         VERIFY0(dsl_deleg_destroy(mos, dd->dd_phys->dd_deleg_zapobj, tx));
671         VERIFY0(zap_remove(mos,
672             dd->dd_parent->dd_phys->dd_child_dir_zapobj, dd->dd_myname, tx));

674         dsl_dir_rele(dd, FTAG);
675         dmu_object_free_zapified(mos, ddobj, tx);
674         VERIFY0(dmu_object_free(mos, ddobj, tx));
676 }

678 void
679 dsl_destroy_head_sync_impl(dsl_dataset_t *ds, dmu_tx_t *tx)
680 {
681         dsl_pool_t *dp = dmu_tx_pool(tx);
682         objset_t *mos = dp->dp_meta_objset;
683         uint64_t obj, ddobj, prevobj = 0;
684         boolean_t rmorigin;

686         ASSERT3U(ds->ds_phys->ds_num_children, <=, 1);
687         ASSERT(ds->ds_prev == NULL ||
```



```
688             ds->ds_prev->ds_phys->ds_next_snap_obj != ds->ds_object);
689         ASSERT3U(ds->ds_phys->ds_bp.blk_birth, <=, tx->tx_txg);
690         ASSERT(RRW_WRITE_HELD(&dp->dp_config_rwlock));

692         /* We need to log before removing it from the namespace. */
693         spa_history_log_internal_ds(ds, "destroy", tx, "");

695         rmorigin = (dsl_dir_is_clone(ds->ds_dir) &&
696             DS_IS_DEFER_DESTROY(ds->ds_prev) &&
697             ds->ds_prev->ds_phys->ds_num_children == 2 &&
698             ds->ds_prev->ds_userrefs == 0);

700         /* Remove our reservation */
701         if (ds->ds_reserved != 0) {
702                 dsl_dataset_set_refreservation_sync_impl(ds,
703                     (ZPROP_SRC_NONE | ZPROP_SRC_LOCAL | ZPROP_SRC_RECEIVED),
704                     0, tx);
705                 ASSERT0(ds->ds_reserved);
706         }

708         dsl_scan_ds_destroyed(ds, tx);

710         obj = ds->ds_object;

712         if (ds->ds_phys->ds_prev_snap_obj != 0) {
713                 /* This is a clone */
714                 ASSERT(ds->ds_prev != NULL);
715                 ASSERT3U(ds->ds_prev->ds_phys->ds_next_snap_obj, !=, obj);
716                 ASSERT0(ds->ds_phys->ds_next_snap_obj);

718                 dmu_buf_will_dirty(ds->ds_prev->ds_dbuf, tx);
719                 if (ds->ds_prev->ds_phys->ds_next_clones_obj != 0) {
720                         dsl_dataset_remove_from_next_clones(ds->ds_prev,
721                             obj, tx);
722                 }

724                 ASSERT3U(ds->ds_prev->ds_phys->ds_num_children, >, 1);
725                 ds->ds_prev->ds_phys->ds_num_children--;
726         }

727         zfeature_info_t *async_destroy =
728             &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY];
729         objset_t *os;

728         /*
729          * Destroy the deadlist.  Unless it's a clone, the
730          * deadlist should be empty.  (If it's a clone, it's
731          * safe to ignore the deadlist contents.)
732          */
733         dsl_deadlist_close(&ds->ds_deadlist);
734         dsl_deadlist_free(mos, ds->ds_phys->ds_deadlist_obj, tx);
735         dmu_buf_will_dirty(ds->ds_dbuf, tx);
736         ds->ds_phys->ds_deadlist_obj = 0;

738         objset_t *os;
739         VERIFY0(dmu_objset_from_ds(ds, &os));

741         if (!spa_feature_is_enabled(dp->dp_spa, SPA_FEATURE_ASYNC_DESTROY)) {
743         if (!spa_feature_is_enabled(dp->dp_spa, async_destroy)) {
742                 old_synchronous_dataset_destroy(ds, tx);
743         } else {
744                 /*
745                  * Move the bptree into the pool's list of trees to
746                  * clean up and update space accounting information.
747                  */
748                 uint64_t used, comp, uncomp;
```

```
750                 zil_destroy_sync(dmu_objset_zil(os), tx);
752                 if (!spa_feature_is_active(dp->dp_spa,
753                     SPA_FEATURE_ASYNC_DESTROY)) {
754                 if (!spa_feature_is_active(dp->dp_spa, async_destroy)) {
754                         dsl_scan_t *scn = dp->dp_scan;
755                         spa_feature_incr(dp->dp_spa, SPA_FEATURE_ASYNC_DESTROY,
756                             tx);
757                         spa_feature_incr(dp->dp_spa, async_destroy, tx);
757                         dp->dp_bptree_obj = bptree_alloc(mos, tx);
758                         VERIFY0(zap_add(mos,
759                             DMU_POOL_DIRECTORY_OBJECT,
760                             DMU_POOL_BPTREE_OBJ, sizeof (uint64_t), 1,
761                             &dp->dp_bptree_obj, tx));
762                         ASSERT(!scn->scn_async_destroying);
763                         scn->scn_async_destroying = B_TRUE;
764                 }
766                 used = ds->ds_dir->dd_phys->dd_used_bytes;
767                 comp = ds->ds_dir->dd_phys->dd_compressed_bytes;
768                 uncomp = ds->ds_dir->dd_phys->dd_uncompressed_bytes;
770                 ASSERT(!DS_UNIQUE_IS_ACCURATE(ds) ||
771                     ds->ds_phys->ds_unique_bytes == used);
773                 bptree_add(mos, dp->dp_bptree_obj,
774                     &ds->ds_phys->ds_bp, ds->ds_phys->ds_prev_snap_txg,
775                     used, comp, uncomp, tx);
776                 dsl_dir_diduse_space(ds->ds_dir, DD_USED_HEAD,
777                     -used, -comp, -uncomp, tx);
778                 dsl_dir_diduse_space(dp->dp_free_dir, DD_USED_HEAD,
779                     used, comp, uncomp, tx);
780         }
782         if (ds->ds_prev != NULL) {
783                 if (spa_version(dp->dp_spa) >= SPA_VERSION_DIR_CLONES) {
784                         VERIFY0(zap_remove_int(mos,
785                             ds->ds_prev->ds_dir->dd_phys->dd_clones,
786                             ds->ds_object, tx));
787                 }
788                 prevobj = ds->ds_prev->ds_object;
789                 dsl_dataset_rele(ds->ds_prev, ds);
790                 ds->ds_prev = NULL;
791         }
793         /*
794          * This must be done after the dsl_traverse(), because it will
795          * re-open the objset.
796          */
797         if (ds->ds_objset) {
798                 dmu_objset_evict(ds->ds_objset);
799                 ds->ds_objset = NULL;
800         }
802         /* Erase the link in the dir */
803         dmu_buf_will_dirty(ds->ds_dir->dd_dbuf, tx);
804         ds->ds_dir->dd_phys->dd_head_dataset_obj = 0;
805         ddobj = ds->ds_dir->dd_object;
806         ASSERT(ds->ds_phys->ds_snapnames_zapobj != 0);
807         VERIFY0(zap_destroy(mos, ds->ds_phys->ds_snapnames_zapobj, tx));
809         spa_prop_clear_bootfs(dp->dp_spa, ds->ds_object, tx);
811         ASSERT0(ds->ds_phys->ds_next_clones_obj);
```



```
812         ASSERT0(ds->ds_phys->ds_props_obj);
813         ASSERT0(ds->ds_phys->ds_userrefs_obj);
814         dsl_dir_rele(ds->ds_dir, ds);
815         ds->ds_dir = NULL;
816         dmu_object_free_zapified(mos, obj, tx);
817         VERIFY0(dmu_object_free(mos, obj, tx));
818         dsl_dir_destroy_sync(ddobj, tx);
820         if (rmorigin) {
821                 dsl_dataset_t *prev;
822                 VERIFY0(dsl_dataset_hold_obj(dp, prevobj, FTAG, &prev));
823                 dsl_destroy_snapshot_sync_impl(prev, B_FALSE, tx);
824                 dsl_dataset_rele(prev, FTAG);
825         }
826 }
_____unchanged_portion_omitted_
857 int
858 dsl_destroy_head(const char *name)
859 {
860         dsl_destroy_head_arg_t ddha;
861         int error;
862         spa_t *spa;
863         boolean_t isenabled;
865 #ifdef _KERNEL
866         zfs_destroy_unmount_origin(name);
867 #endif
869         error = spa_open(name, &spa, FTAG);
870         if (error != 0)
871                 return (error);
872         isenabled = spa_feature_is_enabled(spa, SPA_FEATURE_ASYNC_DESTROY);
873         isenabled = spa_feature_is_enabled(spa,
874             &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY]);
873         spa_close(spa, FTAG);
875         ddha.ddha_name = name;
877         if (!isenabled) {
878                 objset_t *os;
880                 error = dsl_sync_task(name, dsl_destroy_head_check,
881                     dsl_destroy_head_begin_sync, &ddha, 0);
882                 if (error != 0)
883                         return (error);
885                 /*
886                  * Head deletion is processed in one txg on old pools;
887                  * remove the objects from open context so that the txg sync
888                  * is not too long.
889                  */
890                 error = dmu_objset_own(name, DMU_OST_ANY, B_FALSE, FTAG, &os);
891                 if (error == 0) {
892                         uint64_t prev_snap_txg =
893                             dmu_objset_ds(os)->ds_phys->ds_prev_snap_txg;
894                         for (uint64_t obj = 0; error == 0;
895                             error = dmu_object_next(os, &obj, FALSE,
896                             prev_snap_txg))
897                                 (void) dmu_free_long_object(os, obj);
898                         /* sync out all frees */
899                         txg_wait_synced(dmu_objset_pool(os), 0);
900                         dmu_objset_disown(os, FTAG);
901                 }
902         }
```

```
 904         return (dsl_sync_task(name, dsl_destroy_head_check,
 905             dsl_destroy_head_sync, &ddha, 0));
 906 }
```

*unchanged_portion_omitted_*

```
**********************************************************
   35887 Tue Oct  1 14:04:35 2013
new/usr/src/uts/common/fs/zfs/dsl_dir.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2013 Martin Matuska. All rights reserved.
 */

#include <sys/dmu.h>
#include <sys/dmu_objset.h>
#include <sys/dmu_tx.h>
#include <sys/dsl_dataset.h>
#include <sys/dsl_dir.h>
#include <sys/dsl_prop.h>
#include <sys/dsl_synctask.h>
#include <sys/dsl_deleg.h>
#include <sys/dmu_impl.h>
#include <sys/spa.h>
#include <sys/metaslab.h>
#include <sys/zap.h>
#include <sys/zio.h>
#include <sys/arc.h>
#include <sys/sunddi.h>
#include "zfs_namecheck.h"

static uint64_t dsl_dir_space_towrite(dsl_dir_t *dd);

/* ARGSUSED */
static void
dsl_dir_evict(dmu_buf_t *db, void *arg)
{
        dsl_dir_t *dd = arg;
        dsl_pool_t *dp = dd->dd_pool;
        int t;

        for (t = 0; t < TXG_SIZE; t++) {
                ASSERT(!txg_list_member(&dp->dp_dirty_dirs, dd, t));
                ASSERT(dd->dd_tempreserved[t] == 0);
                ASSERT(dd->dd_space_towrite[t] == 0);
        }

        if (dd->dd_parent)
                dsl_dir_rele(dd->dd_parent, dd);

        spa_close(dd->dd_pool->dp_spa, dd);

        /*
         * The props callback list should have been cleaned up by
         * objset_evict().
         */
        list_destroy(&dd->dd_prop_cbs);
        mutex_destroy(&dd->dd_lock);
        kmem_free(dd, sizeof (dsl_dir_t));
}

int
dsl_dir_hold_obj(dsl_pool_t *dp, uint64_t ddobj,
    const char *tail, void *tag, dsl_dir_t **ddp)
{
        dmu_buf_t *dbuf;
        dsl_dir_t *dd;
        int err;

        ASSERT(dsl_pool_config_held(dp));

        err = dmu_bonus_hold(dp->dp_meta_objset, ddobj, tag, &dbuf);
        if (err != 0)
                return (err);
        dd = dmu_buf_get_user(dbuf);
#ifdef ZFS_DEBUG
        {
                dmu_object_info_t doi;
                dmu_object_info_from_db(dbuf, &doi);
                ASSERT3U(doi.doi_bonus_type, ==, DMU_OT_DSL_DIR);
                ASSERT3U(doi.doi_bonus_size, >=, sizeof (dsl_dir_phys_t));
        }
#endif
        if (dd == NULL) {
                dsl_dir_t *winner;

                dd = kmem_zalloc(sizeof (dsl_dir_t), KM_SLEEP);
                dd->dd_object = ddobj;
                dd->dd_dbuf = dbuf;
                dd->dd_pool = dp;
                dd->dd_phys = dbuf->db_data;
                mutex_init(&dd->dd_lock, NULL, MUTEX_DEFAULT, NULL);

                list_create(&dd->dd_prop_cbs, sizeof (dsl_prop_cb_record_t),
                    offsetof(dsl_prop_cb_record_t, cbr_node));

                dsl_dir_snap_cmtime_update(dd);

                if (dd->dd_phys->dd_parent_obj) {
                        err = dsl_dir_hold_obj(dp, dd->dd_phys->dd_parent_obj,
                            NULL, dd, &dd->dd_parent);
                        if (err != 0)
                                goto errout;
                        if (tail) {
#ifdef ZFS_DEBUG
                                uint64_t foundobj;

                                err = zap_lookup(dp->dp_meta_objset,
                                    dd->dd_parent->dd_phys->dd_child_dir_zapobj,
                                    tail, sizeof (foundobj), 1, &foundobj);
```

```
123                                 ASSERT(err || foundobj == ddobj);
124 #endif
125                                 (void) strcpy(dd->dd_myname, tail);
126                         } else {
127                                 err = zap_value_search(dp->dp_meta_objset,
128                                     dd->dd_parent->dd_phys->dd_child_dir_zapobj,
129                                     ddobj, 0, dd->dd_myname);
130                         }
131                         if (err != 0)
132                                 goto errout;
133                 } else {
134                         (void) strcpy(dd->dd_myname, spa_name(dp->dp_spa));
135                 }

137                 if (dsl_dir_is_clone(dd)) {
138                         dmu_buf_t *origin_bonus;
139                         dsl_dataset_phys_t *origin_phys;

141                         /*
142                          * We can't open the origin dataset, because
143                          * that would require opening this dsl_dir.
144                          * Just look at its phys directly instead.
145                          */
146                         err = dmu_bonus_hold(dp->dp_meta_objset,
147                             dd->dd_phys->dd_origin_obj, FTAG, &origin_bonus);
148                         if (err != 0)
149                                 goto errout;
150                         origin_phys = origin_bonus->db_data;
151                         dd->dd_origin_txg =
152                             origin_phys->ds_creation_txg;
153                         dmu_buf_rele(origin_bonus, FTAG);
154                 }

156                 winner = dmu_buf_set_user_ie(dbuf, dd, &dd->dd_phys,
157                     dsl_dir_evict);
158                 if (winner) {
159                         if (dd->dd_parent)
160                                 dsl_dir_rele(dd->dd_parent, dd);
161                         mutex_destroy(&dd->dd_lock);
162                         kmem_free(dd, sizeof (dsl_dir_t));
163                         dd = winner;
164                 } else {
165                         spa_open_ref(dp->dp_spa, dd);
166                 }
167         }

169         /*
170          * The dsl_dir_t has both open-to-close and instantiate-to-evict
171          * holds on the spa.  We need the open-to-close holds because
172          * otherwise the spa_refcnt wouldn't change when we open a
173          * dir which the spa also has open, so we could incorrectly
174          * think it was OK to unload/export/destroy the pool.  We need
175          * the instantiate-to-evict hold because the dsl_dir_t has a
176          * pointer to the dd_pool, which has a pointer to the spa_t.
177          */
178         spa_open_ref(dp->dp_spa, tag);
179         ASSERT3P(dd->dd_pool, ==, dp);
180         ASSERT3U(dd->dd_object, ==, ddobj);
181         ASSERT3P(dd->dd_dbuf, ==, dbuf);
182         *ddp = dd;
183         return (0);

185 errout:
186         if (dd->dd_parent)
187                 dsl_dir_rele(dd->dd_parent, dd);
188         mutex_destroy(&dd->dd_lock);
```



```
189         kmem_free(dd, sizeof (dsl_dir_t));
190         dmu_buf_rele(dbuf, tag);
191         return (err);
192 }
_____unchanged_portion_omitted_

1346 void
1347 dsl_dir_snap_cmtime_update(dsl_dir_t *dd)
1348 {
1349         timestruc_t t;

1351         gethrestime(&t);
1352         mutex_enter(&dd->dd_lock);
1353         dd->dd_snap_cmtime = t;
1354         mutex_exit(&dd->dd_lock);
1355 }

1357 void
1358 dsl_dir_zapify(dsl_dir_t *dd, dmu_tx_t *tx)
1359 {
1360         objset_t *mos = dd->dd_pool->dp_meta_objset;
1361         dmu_object_zapify(mos, dd->dd_object, DMU_OT_DSL_DIR, tx);
1362 }
_____unchanged_portion_omitted_
```

```
**********************************************************
   30598 Tue Oct  1 14:04:35 2013
new/usr/src/uts/common/fs/zfs/dsl_pool.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 198 int
 199 dsl_pool_open(dsl_pool_t *dp)
 200 {
 201         int err;
 202         dsl_dir_t *dd;
 203         dsl_dataset_t *ds;
 204         uint64_t obj;

 206         rrw_enter(&dp->dp_config_rwlock, RW_WRITER, FTAG);
 207         err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 208             DMU_POOL_ROOT_DATASET, sizeof (uint64_t), 1,
 209             &dp->dp_root_dir_obj);
 210         if (err)
 211                 goto out;

 213         err = dsl_dir_hold_obj(dp, dp->dp_root_dir_obj,
 214             NULL, dp, &dp->dp_root_dir);
 215         if (err)
 216                 goto out;

 218         err = dsl_pool_open_special_dir(dp, MOS_DIR_NAME, &dp->dp_mos_dir);
 219         if (err)
 220                 goto out;

 222         if (spa_version(dp->dp_spa) >= SPA_VERSION_ORIGIN) {
 223                 err = dsl_pool_open_special_dir(dp, ORIGIN_DIR_NAME, &dd);
 224                 if (err)
 225                         goto out;
 226                 err = dsl_dataset_hold_obj(dp, dd->dd_phys->dd_head_dataset_obj,
 227                     FTAG, &ds);
 228                 if (err == 0) {
 229                         err = dsl_dataset_hold_obj(dp,
 230                             ds->ds_phys->ds_prev_snap_obj, dp,
 231                             &dp->dp_origin_snap);
 232                         dsl_dataset_rele(ds, FTAG);
 233                 }
 234                 dsl_dir_rele(dd, dp);
 235                 if (err)
 236                         goto out;
 237         }

 239         if (spa_version(dp->dp_spa) >= SPA_VERSION_DEADLISTS) {
 240                 err = dsl_pool_open_special_dir(dp, FREE_DIR_NAME,
 241                     &dp->dp_free_dir);
 242                 if (err)
 243                         goto out;

 245                 err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 246                     DMU_POOL_FREE_BPOBJ, sizeof (uint64_t), 1, &obj);
 247                 if (err)
 248                         goto out;
 249                 VERIFY0(bpobj_open(&dp->dp_free_bpobj,
 250                     dp->dp_meta_objset, obj));
 251         }
```



```
 253         if (spa_feature_is_active(dp->dp_spa, SPA_FEATURE_ASYNC_DESTROY)) {
 253         if (spa_feature_is_active(dp->dp_spa,
 254             &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
 254                 err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 255                     DMU_POOL_BPTREE_OBJ, sizeof (uint64_t), 1,
 256                     &dp->dp_bptree_obj);
 257                 if (err != 0)
 258                         goto out;
 259         }

 261         if (spa_feature_is_active(dp->dp_spa, SPA_FEATURE_EMPTY_BPOBJ)) {
 262         if (spa_feature_is_active(dp->dp_spa,
 263             &spa_feature_table[SPA_FEATURE_EMPTY_BPOBJ])) {
 262                 err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 263                     DMU_POOL_EMPTY_BPOBJ, sizeof (uint64_t), 1,
 264                     &dp->dp_empty_bpobj);
 265                 if (err != 0)
 266                         goto out;
 267         }

 269         err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 270             DMU_POOL_TMP_USERREFS, sizeof (uint64_t), 1,
 271             &dp->dp_tmp_userrefs_obj);
 272         if (err == ENOENT)
 273                 err = 0;
 274         if (err)
 275                 goto out;

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

 279 out:
 280         rrw_exit(&dp->dp_config_rwlock, FTAG);
 281         return (err);
 282 }
______unchanged_portion_omitted_
```

```
**********************************************************
   50297 Tue Oct  1 14:04:37 2013
new/usr/src/uts/common/fs/zfs/dsl_scan.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

  87 int
  88 dsl_scan_init(dsl_pool_t *dp, uint64_t txg)
  89 {
  90         int err;
  91         dsl_scan_t *scn;
  92         spa_t *spa = dp->dp_spa;
  93         uint64_t f;

  95         scn = dp->dp_scan = kmem_zalloc(sizeof (dsl_scan_t), KM_SLEEP);
  96         scn->scn_dp = dp;

  98         /*
  99          * It's possible that we're resuming a scan after a reboot so
 100          * make sure that the scan_async_destroying flag is initialized
 101          * appropriately.
 102          */
 103         ASSERT(!scn->scn_async_destroying);
 104         scn->scn_async_destroying = spa_feature_is_active(dp->dp_spa,
 105             SPA_FEATURE_ASYNC_DESTROY);
 105             &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY]);

 107         err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 108             "scrub_func", sizeof (uint64_t), 1, &f);
 109         if (err == 0) {
 110                 /*
 111                  * There was an old-style scrub in progress.  Restart a
 112                  * new-style scrub from the beginning.
 113                  */
 114                 scn->scn_restart_txg = txg;
 115                 zfs_dbgmsg("old-style scrub was in progress; "
 116                     "restarting new-style scrub in txg %llu",
 117                     scn->scn_restart_txg);

 119                 /*
 120                  * Load the queue obj from the old location so that it
 121                  * can be freed by dsl_scan_done().
 122                  */
 123                 (void) zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 124                     "scrub_queue", sizeof (uint64_t), 1,
 125                     &scn->scn_phys.scn_queue_obj);
 126         } else {
 127                 err = zap_lookup(dp->dp_meta_objset, DMU_POOL_DIRECTORY_OBJECT,
 128                     DMU_POOL_SCAN, sizeof (uint64_t), SCAN_PHYS_NUMINTS,
 129                     &scn->scn_phys);
 130                 if (err == ENOENT)
 131                         return (0);
 132                 else if (err)
 133                         return (err);

 135                 if (scn->scn_phys.scn_state == DSS_SCANNING &&
 136                     spa_prev_software_version(dp->dp_spa) < SPA_VERSION_SCAN) {
 137                         /*
 138                          * A new-type scrub was in progress on an old
 139                          * pool, and the pool was accessed by old
 140                          * software.  Restart from the beginning, since
```



```
 141                          * the old software may have changed the pool in
 142                          * the meantime.
 143                          */
 144                         scn->scn_restart_txg = txg;
 145                         zfs_dbgmsg("new-style scrub was modified "
 146                             "by old software; restarting in txg %llu",
 147                             scn->scn_restart_txg);
 148                 }
 149         }

 151         spa_scan_stat_init(spa);
 152         return (0);
 153 }
______unchanged_portion_omitted_

1346 boolean_t
1347 dsl_scan_active(dsl_scan_t *scn)
1348 {
1349         spa_t *spa = scn->scn_dp->dp_spa;
1350         uint64_t used = 0, comp, uncomp;

1352         if (spa->spa_load_state != SPA_LOAD_NONE)
1353                 return (B_FALSE);
1354         if (spa_shutting_down(spa))
1355                 return (B_FALSE);

1356         if (scn->scn_phys.scn_state == DSS_SCANNING ||
1357             scn->scn_async_destroying)
1358                 return (B_TRUE);

1360         if (spa_version(scn->scn_dp->dp_spa) >= SPA_VERSION_DEADLISTS) {
1361                 (void) bpobj_space(&scn->scn_dp->dp_free_bpobj,
1362                     &used, &comp, &uncomp);
1363         }
1364         return (used != 0);
1365 }

1367 void
1368 dsl_scan_sync(dsl_pool_t *dp, dmu_tx_t *tx)
1369 {
1370         dsl_scan_t *scn = dp->dp_scan;
1371         spa_t *spa = dp->dp_spa;
1372         int err;

1374         /*
1375          * Check for scn_restart_txg before checking spa_load_state, so
1376          * that we can restart an old-style scan while the pool is being
1377          * imported (see dsl_scan_init).
1378          */
1379         if (scn->scn_restart_txg != 0 &&
1380             scn->scn_restart_txg <= tx->tx_txg) {
1381                 pool_scan_func_t func = POOL_SCAN_SCRUB;
1382                 dsl_scan_done(scn, B_FALSE, tx);
1383                 if (vdev_resilver_needed(spa->spa_root_vdev, NULL, NULL))
1384                         func = POOL_SCAN_RESILVER;
1385                 zfs_dbgmsg("restarting scan func=%u txg=%llu",
1386                     func, tx->tx_txg);
1387                 dsl_scan_setup_sync(&func, tx);
1388         }

1390         if (!dsl_scan_active(scn) ||
1391             spa_sync_pass(dp->dp_spa) > 1)
1392                 return;

1394         scn->scn_visited_this_txg = 0;
1395         scn->scn_pausing = B_FALSE;
```

```
1396         scn->scn_sync_start_time = gethrtime();
1397         spa->spa_scrub_active = B_TRUE;

1399         /*
1400          * First process the free list.  If we pause the free, don't do
1401          * any scanning.  This ensures that there is no free list when
1402          * we are scanning, so the scan code doesn't have to worry about
1403          * traversing it.
1404          */
1405         if (spa_version(dp->dp_spa) >= SPA_VERSION_DEADLISTS) {
1406                 scn->scn_is_bptree = B_FALSE;
1407                 scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1408                     NULL, ZIO_FLAG_MUSTSUCCEED);
1409                 err = bpobj_iterate(&dp->dp_free_bpobj,
1410                     dsl_scan_free_block_cb, scn, tx);
1411                 VERIFY3U(0, ==, zio_wait(scn->scn_zio_root));

1413                 if (err == 0 && spa_feature_is_active(spa,
1414                     SPA_FEATURE_ASYNC_DESTROY)) {
1415                     &spa_feature_table[SPA_FEATURE_ASYNC_DESTROY])) {
1415                         ASSERT(scn->scn_async_destroying);
1416                         scn->scn_is_bptree = B_TRUE;
1417                         scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1418                             NULL, ZIO_FLAG_MUSTSUCCEED);
1419                         err = bptree_iterate(dp->dp_meta_objset,
1420                             dp->dp_bptree_obj, B_TRUE, dsl_scan_free_block_cb,
1421                             scn, tx);
1422                         VERIFY0(zio_wait(scn->scn_zio_root));

1424                         if (err == 0) {
1426                                 zfeature_info_t *feat = &spa_feature_table
1427                                     [SPA_FEATURE_ASYNC_DESTROY];
1425                                 /* finished; deactivate async destroy feature */
1426                                 spa_feature_decr(spa, SPA_FEATURE_ASYNC_DESTROY,
1427                                     tx);
1428                                 ASSERT(!spa_feature_is_active(spa,
1429                                     SPA_FEATURE_ASYNC_DESTROY));
1429                                 spa_feature_decr(spa, feat, tx);
1430                                 ASSERT(!spa_feature_is_active(spa, feat));
1430                                 VERIFY0(zap_remove(dp->dp_meta_objset,
1431                                     DMU_POOL_DIRECTORY_OBJECT,
1432                                     DMU_POOL_BPTREE_OBJ, tx));
1433                                 VERIFY0(bptree_free(dp->dp_meta_objset,
1434                                     dp->dp_bptree_obj, tx));
1435                                 dp->dp_bptree_obj = 0;
1436                                 scn->scn_async_destroying = B_FALSE;
1437                         }
1438                 }
1439                 if (scn->scn_visited_this_txg) {
1440                         zfs_dbgmsg("freed %llu blocks in %llums from "
1441                             "free_bpobj/bptree txg %llu",
1442                             (longlong_t)scn->scn_visited_this_txg,
1443                             (longlong_t)
1444                             NSEC2MSEC(gethrtime() - scn->scn_sync_start_time),
1445                             (longlong_t)tx->tx_txg);
1446                         scn->scn_visited_this_txg = 0;
1447                         /*
1448                          * Re-sync the ddt so that we can further modify
1449                          * it when doing bprewrite.
1450                          */
1451                         ddt_sync(spa, tx->tx_txg);
1452                 }
1453                 if (err == ERESTART)
1454                         return;
1455         }
```



```
1457         if (scn->scn_phys.scn_state != DSS_SCANNING)
1458                 return;

1460         if (scn->scn_done_txg == tx->tx_txg) {
1461                 ASSERT(!scn->scn_pausing);
1462                 /* finished with scan. */
1463                 zfs_dbgmsg("txg %llu scan complete", tx->tx_txg);
1464                 dsl_scan_done(scn, B_TRUE, tx);
1465                 ASSERT3U(spa->spa_scrub_inflight, ==, 0);
1466                 dsl_scan_sync_state(scn, tx);
1467                 return;
1468         }

1470         if (scn->scn_phys.scn_ddt_bookmark.ddb_class <=
1471             scn->scn_phys.scn_ddt_class_max) {
1472                 zfs_dbgmsg("doing scan sync txg %llu; "
1473                     "ddt bm=%llu/%llu/%llu/%llx",
1474                     (longlong_t)tx->tx_txg,
1475                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_class,
1476                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_type,
1477                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_checksum,
1478                     (longlong_t)scn->scn_phys.scn_ddt_bookmark.ddb_cursor);
1479                 ASSERT(scn->scn_phys.scn_bookmark.zb_objset == 0);
1480                 ASSERT(scn->scn_phys.scn_bookmark.zb_object == 0);
1481                 ASSERT(scn->scn_phys.scn_bookmark.zb_level == 0);
1482                 ASSERT(scn->scn_phys.scn_bookmark.zb_blkid == 0);
1483         } else {
1484                 zfs_dbgmsg("doing scan sync txg %llu; bm=%llu/%llu/%llu/%llu",
1485                     (longlong_t)tx->tx_txg,
1486                     (longlong_t)scn->scn_phys.scn_bookmark.zb_objset,
1487                     (longlong_t)scn->scn_phys.scn_bookmark.zb_object,
1488                     (longlong_t)scn->scn_phys.scn_bookmark.zb_level,
1489                     (longlong_t)scn->scn_phys.scn_bookmark.zb_blkid);
1490         }

1492         scn->scn_zio_root = zio_root(dp->dp_spa, NULL,
1493             NULL, ZIO_FLAG_CANFAIL);
1494         dsl_pool_config_enter(dp, FTAG);
1495         dsl_scan_visit(scn, tx);
1496         dsl_pool_config_exit(dp, FTAG);
1497         (void) zio_wait(scn->scn_zio_root);
1498         scn->scn_zio_root = NULL;

1500         zfs_dbgmsg("visited %llu blocks in %llums",
1501             (longlong_t)scn->scn_visited_this_txg,
1502             (longlong_t)NSEC2MSEC(gethrtime() - scn->scn_sync_start_time));

1504         if (!scn->scn_pausing) {
1505                 scn->scn_done_txg = tx->tx_txg + 1;
1506                 zfs_dbgmsg("txg %llu traversal complete, waiting till txg %llu",
1507                     tx->tx_txg, scn->scn_done_txg);
1508         }

1510         if (DSL_SCAN_IS_SCRUB_RESILVER(scn)) {
1511                 mutex_enter(&spa->spa_scrub_lock);
1512                 while (spa->spa_scrub_inflight > 0) {
1513                         cv_wait(&spa->spa_scrub_io_cv,
1514                             &spa->spa_scrub_lock);
1515                 }
1516                 mutex_exit(&spa->spa_scrub_lock);
1517         }

1519         dsl_scan_sync_state(scn, tx);
1520 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
  175806 Tue Oct  1 14:04:39 2013
new/usr/src/uts/common/fs/zfs/spa.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

2070 /*
2071  * Load an existing storage pool, using the pool's builtin spa_config as a
2072  * source of configuration information.
2073  */
2074 static int
2075 spa_load_impl(spa_t *spa, uint64_t pool_guid, nvlist_t *config,
2076     spa_load_state_t state, spa_import_type_t type, boolean_t mosconfig,
2077     char **ereport)
2078 {
2079         int error = 0;
2080         nvlist_t *nvroot = NULL;
2081         nvlist_t *label;
2082         vdev_t *rvd;
2083         uberblock_t *ub = &spa->spa_uberblock;
2084         uint64_t children, config_cache_txg = spa->spa_config_txg;
2085         int orig_mode = spa->spa_mode;
2086         int parse;
2087         uint64_t obj;
2088         boolean_t missing_feat_write = B_FALSE;

2090         /*
2091          * If this is an untrusted config, access the pool in read-only mode.
2092          * This prevents things like resilvering recently removed devices.
2093          */
2094         if (!mosconfig)
2095                 spa->spa_mode = FREAD;

2097         ASSERT(MUTEX_HELD(&spa_namespace_lock));

2099         spa->spa_load_state = state;

2101         if (nvlist_lookup_nvlist(config, ZPOOL_CONFIG_VDEV_TREE, &nvroot))
2102                 return (SET_ERROR(EINVAL));

2104         parse = (type == SPA_IMPORT_EXISTING ?
2105             VDEV_ALLOC_LOAD : VDEV_ALLOC_SPLIT);

2107         /*
2108          * Create "The Godfather" zio to hold all async IOs
2109          */
2110         spa->spa_async_zio_root = zio_root(spa, NULL, NULL,
2111             ZIO_FLAG_CANFAIL | ZIO_FLAG_SPECULATIVE | ZIO_FLAG_GODFATHER);

2113         /*
2114          * Parse the configuration into a vdev tree.  We explicitly set the
2115          * value that will be returned by spa_version() since parsing the
2116          * configuration requires knowing the version number.
2117          */
2118         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2119         error = spa_config_parse(spa, &rvd, nvroot, NULL, 0, parse);
2120         spa_config_exit(spa, SCL_ALL, FTAG);

2122         if (error != 0)
2123                 return (error);
```



```
2125         ASSERT(spa->spa_root_vdev == rvd);

2127         if (type != SPA_IMPORT_ASSEMBLE) {
2128                 ASSERT(spa_guid(spa) == pool_guid);
2129         }

2131         /*
2132          * Try to open all vdevs, loading each label in the process.
2133          */
2134         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2135         error = vdev_open(rvd);
2136         spa_config_exit(spa, SCL_ALL, FTAG);
2137         if (error != 0)
2138                 return (error);

2140         /*
2141          * We need to validate the vdev labels against the configuration that
2142          * we have in hand, which is dependent on the setting of mosconfig. If
2143          * mosconfig is true then we're validating the vdev labels based on
2144          * that config.  Otherwise, we're validating against the cached config
2145          * (zpool.cache) that was read when we loaded the zfs module, and then
2146          * later we will recursively call spa_load() and validate against
2147          * the vdev config.
2148          *
2149          * If we're assembling a new pool that's been split off from an
2150          * existing pool, the labels haven't yet been updated so we skip
2151          * validation for now.
2152          */
2153         if (type != SPA_IMPORT_ASSEMBLE) {
2154                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2155                 error = vdev_validate(rvd, mosconfig);
2156                 spa_config_exit(spa, SCL_ALL, FTAG);

2158                 if (error != 0)
2159                         return (error);

2161                 if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2162                         return (SET_ERROR(ENXIO));
2163         }

2165         /*
2166          * Find the best uberblock.
2167          */
2168         vdev_uberblock_load(rvd, ub, &label);

2170         /*
2171          * If we weren't able to find a single valid uberblock, return failure.
2172          */
2173         if (ub->ub_txg == 0) {
2174                 nvlist_free(label);
2175                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, ENXIO));
2176         }

2178         /*
2179          * If the pool has an unsupported version we can't open it.
2180          */
2181         if (!SPA_VERSION_IS_SUPPORTED(ub->ub_version)) {
2182                 nvlist_free(label);
2183                 return (spa_vdev_err(rvd, VDEV_AUX_VERSION_NEWER, ENOTSUP));
2184         }

2186         if (ub->ub_version >= SPA_VERSION_FEATURES) {
2187                 nvlist_t *features;

2189                 /*
2190                  * If we weren't able to find what's necessary for reading the
```

```
2191                 * MOS in the label, return failure.
2192                 */
2193                if (label == NULL || nvlist_lookup_nvlist(label,
2194                    ZPOOL_CONFIG_FEATURES_FOR_READ, &features) != 0) {
2195                        nvlist_free(label);
2196                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2197                            ENXIO));
2198                }

2200                /*
2201                 * Update our in-core representation with the definitive values
2202                 * from the label.
2203                 */
2204                nvlist_free(spa->spa_label_features);
2205                VERIFY(nvlist_dup(features, &spa->spa_label_features, 0) == 0);
2206        }

2208        nvlist_free(label);

2210        /*
2211         * Look through entries in the label nvlist's features_for_read. If
2212         * there is a feature listed there which we don't understand then we
2213         * cannot open a pool.
2214         */
2215        if (ub->ub_version >= SPA_VERSION_FEATURES) {
2216                nvlist_t *unsup_feat;

2218                VERIFY(nvlist_alloc(&unsup_feat, NV_UNIQUE_NAME, KM_SLEEP) ==
2219                    0);

2221                for (nvpair_t *nvp = nvlist_next_nvpair(spa->spa_label_features,
2222                    NULL); nvp != NULL;
2223                    nvp = nvlist_next_nvpair(spa->spa_label_features, nvp)) {
2224                        if (!zfeature_is_supported(nvpair_name(nvp))) {
2225                                VERIFY(nvlist_add_string(unsup_feat,
2226                                    nvpair_name(nvp), "") == 0);
2227                        }
2228                }

2230                if (!nvlist_empty(unsup_feat)) {
2231                        VERIFY(nvlist_add_nvlist(spa->spa_load_info,
2232                            ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat) == 0);
2233                        nvlist_free(unsup_feat);
2234                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2235                            ENOTSUP));
2236                }

2238                nvlist_free(unsup_feat);
2239        }

2241        /*
2242         * If the vdev guid sum doesn't match the uberblock, we have an
2243         * incomplete configuration.  We first check to see if the pool
2244         * is aware of the complete config (i.e ZPOOL_CONFIG_VDEV_CHILDREN).
2245         * If it is, defer the vdev_guid_sum check till later so we
2246         * can handle missing vdevs.
2247         */
2248        if (nvlist_lookup_uint64(config, ZPOOL_CONFIG_VDEV_CHILDREN,
2249            &children) != 0 && mosconfig && type != SPA_IMPORT_ASSEMBLE &&
2250            rvd->vdev_guid_sum != ub->ub_guid_sum)
2251                return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM, ENXIO));

2253        if (type != SPA_IMPORT_ASSEMBLE && spa->spa_config_splitting) {
2254                spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2255                spa_try_repair(spa, config);
2256                spa_config_exit(spa, SCL_ALL, FTAG);
```



```
2257                nvlist_free(spa->spa_config_splitting);
2258                spa->spa_config_splitting = NULL;
2259        }

2261        /*
2262         * Initialize internal SPA structures.
2263         */
2264        spa->spa_state = POOL_STATE_ACTIVE;
2265        spa->spa_ubsync = spa->spa_uberblock;
2266        spa->spa_verify_min_txg = spa->spa_extreme_rewind ?
2267            TXG_INITIAL - 1 : spa_last_synced_txg(spa) - TXG_DEFER_SIZE - 1;
2268        spa->spa_first_txg = spa->spa_last_ubsync_txg ?
2269            spa->spa_last_ubsync_txg : spa_last_synced_txg(spa) + 1;
2270        spa->spa_claim_max_txg = spa->spa_first_txg;
2271        spa->spa_prev_software_version = ub->ub_software_version;

2273        error = dsl_pool_init(spa, spa->spa_first_txg, &spa->spa_dsl_pool);
2274        if (error)
2275                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2276        spa->spa_meta_objset = spa->spa_dsl_pool->dp_meta_objset;

2278        if (spa_dir_prop(spa, DMU_POOL_CONFIG, &spa->spa_config_object) != 0)
2279                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));

2281        if (spa_version(spa) >= SPA_VERSION_FEATURES) {
2282                boolean_t missing_feat_read = B_FALSE;
2283                nvlist_t *unsup_feat, *enabled_feat;

2285                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_READ,
2286                    &spa->spa_feat_for_read_obj) != 0) {
2287                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2288                }

2290                if (spa_dir_prop(spa, DMU_POOL_FEATURES_FOR_WRITE,
2291                    &spa->spa_feat_for_write_obj) != 0) {
2292                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2293                }

2295                if (spa_dir_prop(spa, DMU_POOL_FEATURE_DESCRIPTIONS,
2296                    &spa->spa_feat_desc_obj) != 0) {
2297                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2298                }

2300                enabled_feat = fnvlist_alloc();
2301                unsup_feat = fnvlist_alloc();

2303                if (!spa_features_check(spa, B_FALSE,
2303                    if (!feature_is_supported(spa->spa_meta_objset,
2304                        spa->spa_feat_for_read_obj, spa->spa_feat_desc_obj,
2304                        unsup_feat, enabled_feat))
2305                                missing_feat_read = B_TRUE;

2307                if (spa_writeable(spa) || state == SPA_LOAD_TRYIMPORT) {
2308                        if (!spa_features_check(spa, B_TRUE,
2309                        if (!feature_is_supported(spa->spa_meta_objset,
2310                            spa->spa_feat_for_write_obj, spa->spa_feat_desc_obj,
2309                            unsup_feat, enabled_feat)) {
2310                                missing_feat_write = B_TRUE;
2311                        }
2312                }

2314                fnvlist_add_nvlist(spa->spa_load_info,
2315                    ZPOOL_CONFIG_ENABLED_FEAT, enabled_feat);

2317                if (!nvlist_empty(unsup_feat)) {
2318                        fnvlist_add_nvlist(spa->spa_load_info,
```

```
2319                                ZPOOL_CONFIG_UNSUP_FEAT, unsup_feat);
2320                }
2321
2322                fnvlist_free(enabled_feat);
2323                fnvlist_free(unsup_feat);
2324
2325                if (!missing_feat_read) {
2326                        fnvlist_add_boolean(spa->spa_load_info,
2327                            ZPOOL_CONFIG_CAN_RDONLY);
2328                }
2329
2330                /*
2331                 * If the state is SPA_LOAD_TRYIMPORT, our objective is
2332                 * twofold: to determine whether the pool is available for
2333                 * import in read-write mode and (if it is not) whether the
2334                 * pool is available for import in read-only mode. If the pool
2335                 * is available for import in read-write mode, it is displayed
2336                 * as available in userland; if it is not available for import
2337                 * in read-only mode, it is displayed as unavailable in
2338                 * userland. If the pool is available for import in read-only
2339                 * mode but not read-write mode, it is displayed as unavailable
2340                 * in userland with a special note that the pool is actually
2341                 * available for open in read-only mode.
2342                 *
2343                 * As a result, if the state is SPA_LOAD_TRYIMPORT and we are
2344                 * missing a feature for write, we must first determine whether
2345                 * the pool can be opened read-only before returning to
2346                 * userland in order to know whether to display the
2347                 * abovementioned note.
2348                 */
2349                if (missing_feat_read || (missing_feat_write &&
2350                    spa_writeable(spa))) {
2351                        return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT,
2352                            ENOTSUP));
2353                }
2354        }
2355
2356        spa->spa_is_initializing = B_TRUE;
2357        error = dsl_pool_open(spa->spa_dsl_pool);
2358        spa->spa_is_initializing = B_FALSE;
2359        if (error != 0)
2360                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2361
2362        if (!mosconfig) {
2363                uint64_t hostid;
2364                nvlist_t *policy = NULL, *nvconfig;
2365
2366                if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2367                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2368
2369                if (!spa_is_root(spa) && nvlist_lookup_uint64(nvconfig,
2370                    ZPOOL_CONFIG_HOSTID, &hostid) == 0) {
2371                        char *hostname;
2372                        unsigned long myhostid = 0;
2373
2374                        VERIFY(nvlist_lookup_string(nvconfig,
2375                            ZPOOL_CONFIG_HOSTNAME, &hostname) == 0);
2376
2377 #ifdef  _KERNEL
2378                        myhostid = zone_get_hostid(NULL);
2379 #else   /* _KERNEL */
2380                        /*
2381                         * We're emulating the system's hostid in userland, so
2382                         * we can't use zone_get_hostid().
2383                         */
2384                        (void) ddi_strtoul(hw_serial, NULL, 10, &myhostid);
```



```
2385 #endif  /* _KERNEL */
2386                        if (hostid != 0 && myhostid != 0 &&
2387                            hostid != myhostid) {
2388                                nvlist_free(nvconfig);
2389                                cmn_err(CE_WARN, "pool '%s' could not be "
2390                                    "loaded as it was last accessed by "
2391                                    "another system (host: %s hostid: 0x%lx). "
2392                                    "See: http://illumos.org/msg/ZFS-8000-EY",
2393                                    spa_name(spa), hostname,
2394                                    (unsigned long)hostid);
2395                                return (SET_ERROR(EBADF));
2396                        }
2397                }
2398                if (nvlist_lookup_nvlist(spa->spa_config,
2399                    ZPOOL_REWIND_POLICY, &policy) == 0)
2400                        VERIFY(nvlist_add_nvlist(nvconfig,
2401                            ZPOOL_REWIND_POLICY, policy) == 0);
2402
2403                spa_config_set(spa, nvconfig);
2404                spa_unload(spa);
2405                spa_deactivate(spa);
2406                spa_activate(spa, orig_mode);
2407
2408                return (spa_load(spa, state, SPA_IMPORT_EXISTING, B_TRUE));
2409        }
2410
2411        if (spa_dir_prop(spa, DMU_POOL_SYNC_BPOBJ, &obj) != 0)
2412                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2413        error = bpobj_open(&spa->spa_deferred_bpobj, spa->spa_meta_objset, obj);
2414        if (error != 0)
2415                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2416
2417        /*
2418         * Load the bit that tells us to use the new accounting function
2419         * (raid-z deflation).  If we have an older pool, this will not
2420         * be present.
2421         */
2422        error = spa_dir_prop(spa, DMU_POOL_DEFLATE, &spa->spa_deflate);
2423        if (error != 0 && error != ENOENT)
2424                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2425
2426        error = spa_dir_prop(spa, DMU_POOL_CREATION_VERSION,
2427            &spa->spa_creation_version);
2428        if (error != 0 && error != ENOENT)
2429                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2430
2431        /*
2432         * Load the persistent error log.  If we have an older pool, this will
2433         * not be present.
2434         */
2435        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_LAST, &spa->spa_errlog_last);
2436        if (error != 0 && error != ENOENT)
2437                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2438
2439        error = spa_dir_prop(spa, DMU_POOL_ERRLOG_SCRUB,
2440            &spa->spa_errlog_scrub);
2441        if (error != 0 && error != ENOENT)
2442                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2443
2444        /*
2445         * Load the history object.  If we have an older pool, this
2446         * will not be present.
2447         */
2448        error = spa_dir_prop(spa, DMU_POOL_HISTORY, &spa->spa_history);
2449        if (error != 0 && error != ENOENT)
2450                return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
```

```
2452         /*
2453          * If we're assembling the pool from the split-off vdevs of
2454          * an existing pool, we don't want to attach the spares & cache
2455          * devices.
2456          */
2457
2458         /*
2459          * Load any hot spares for this pool.
2460          */
2461         error = spa_dir_prop(spa, DMU_POOL_SPARES, &spa->spa_spares.sav_object);
2462         if (error != 0 && error != ENOENT)
2463                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2464         if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2465                 ASSERT(spa_version(spa) >= SPA_VERSION_SPARES);
2466                 if (load_nvlist(spa, spa->spa_spares.sav_object,
2467                     &spa->spa_spares.sav_config) != 0)
2468                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2469
2470                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2471                 spa_load_spares(spa);
2472                 spa_config_exit(spa, SCL_ALL, FTAG);
2473         } else if (error == 0) {
2474                 spa->spa_spares.sav_sync = B_TRUE;
2475         }
2476
2477         /*
2478          * Load any level 2 ARC devices for this pool.
2479          */
2480         error = spa_dir_prop(spa, DMU_POOL_L2CACHE,
2481             &spa->spa_l2cache.sav_object);
2482         if (error != 0 && error != ENOENT)
2483                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2484         if (error == 0 && type != SPA_IMPORT_ASSEMBLE) {
2485                 ASSERT(spa_version(spa) >= SPA_VERSION_L2CACHE);
2486                 if (load_nvlist(spa, spa->spa_l2cache.sav_object,
2487                     &spa->spa_l2cache.sav_config) != 0)
2488                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2489
2490                 spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2491                 spa_load_l2cache(spa);
2492                 spa_config_exit(spa, SCL_ALL, FTAG);
2493         } else if (error == 0) {
2494                 spa->spa_l2cache.sav_sync = B_TRUE;
2495         }
2496
2497         spa->spa_delegation = zpool_prop_default_numeric(ZPOOL_PROP_DELEGATION);
2498
2499         error = spa_dir_prop(spa, DMU_POOL_PROPS, &spa->spa_pool_props_object);
2500         if (error && error != ENOENT)
2501                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2502
2503         if (error == 0) {
2504                 uint64_t autoreplace;
2505
2506                 spa_prop_find(spa, ZPOOL_PROP_BOOTFS, &spa->spa_bootfs);
2507                 spa_prop_find(spa, ZPOOL_PROP_AUTOREPLACE, &autoreplace);
2508                 spa_prop_find(spa, ZPOOL_PROP_DELEGATION, &spa->spa_delegation);
2509                 spa_prop_find(spa, ZPOOL_PROP_FAILUREMODE, &spa->spa_failmode);
2510                 spa_prop_find(spa, ZPOOL_PROP_AUTOEXPAND, &spa->spa_autoexpand);
2511                 spa_prop_find(spa, ZPOOL_PROP_DEDUPDITTO,
2512                     &spa->spa_dedup_ditto);
2513
2514                 spa->spa_autoreplace = (autoreplace != 0);
2515         }
```



```
2517         /*
2518          * If the 'autoreplace' property is set, then post a resource notifying
2519          * the ZFS DE that it should not issue any faults for unopenable
2520          * devices.  We also iterate over the vdevs, and post a sysevent for any
2521          * unopenable vdevs so that the normal autoreplace handler can take
2522          * over.
2523          */
2524         if (spa->spa_autoreplace && state != SPA_LOAD_TRYIMPORT) {
2525                 spa_check_removed(spa->spa_root_vdev);
2526                 /*
2527                  * For the import case, this is done in spa_import(), because
2528                  * at this point we're using the spare definitions from
2529                  * the MOS config, not necessarily from the userland config.
2530                  */
2531                 if (state != SPA_LOAD_IMPORT) {
2532                         spa_aux_check_removed(&spa->spa_spares);
2533                         spa_aux_check_removed(&spa->spa_l2cache);
2534                 }
2535         }
2536
2537         /*
2538          * Load the vdev state for all toplevel vdevs.
2539          */
2540         vdev_load(rvd);
2541
2542         /*
2543          * Propagate the leaf DTLs we just loaded all the way up the tree.
2544          */
2545         spa_config_enter(spa, SCL_ALL, FTAG, RW_WRITER);
2546         vdev_dtl_reassess(rvd, 0, 0, B_FALSE);
2547         spa_config_exit(spa, SCL_ALL, FTAG);
2548
2549         /*
2550          * Load the DDTs (dedup tables).
2551          */
2552         error = ddt_load(spa);
2553         if (error != 0)
2554                 return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2555
2556         spa_update_dspace(spa);
2557
2558         /*
2559          * Validate the config, using the MOS config to fill in any
2560          * information which might be missing.  If we fail to validate
2561          * the config then declare the pool unfit for use. If we're
2562          * assembling a pool from a split, the log is not transferred
2563          * over.
2564          */
2565         if (type != SPA_IMPORT_ASSEMBLE) {
2566                 nvlist_t *nvconfig;
2567
2568                 if (load_nvlist(spa, spa->spa_config_object, &nvconfig) != 0)
2569                         return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA, EIO));
2570
2571                 if (!spa_config_valid(spa, nvconfig)) {
2572                         nvlist_free(nvconfig);
2573                         return (spa_vdev_err(rvd, VDEV_AUX_BAD_GUID_SUM,
2574                             ENXIO));
2575                 }
2576                 nvlist_free(nvconfig);
2577
2578                 /*
2579                  * Now that we've validated the config, check the state of the
2580                  * root vdev.  If it can't be opened, it indicates one or
2581                  * more toplevel vdevs are faulted.
2582                  */
```

```
2583                if (rvd->vdev_state <= VDEV_STATE_CANT_OPEN)
2584                        return (SET_ERROR(ENXIO));

2586                if (spa_check_logs(spa)) {
2587                        *ereport = FM_EREPORT_ZFS_LOG_REPLAY;
2588                        return (spa_vdev_err(rvd, VDEV_AUX_BAD_LOG, ENXIO));
2589                }
2590        }

2592        if (missing_feat_write) {
2593                ASSERT(state == SPA_LOAD_TRYIMPORT);

2595                /*
2596                 * At this point, we know that we can open the pool in
2597                 * read-only mode but not read-write mode. We now have enough
2598                 * information and can return to userland.
2599                 */
2600                return (spa_vdev_err(rvd, VDEV_AUX_UNSUP_FEAT, ENOTSUP));
2601        }

2603        /*
2604         * We've successfully opened the pool, verify that we're ready
2605         * to start pushing transactions.
2606         */
2607        if (state != SPA_LOAD_TRYIMPORT) {
2608                if (error = spa_load_verify(spa))
2609                        return (spa_vdev_err(rvd, VDEV_AUX_CORRUPT_DATA,
2610                            error));
2611        }

2613        if (spa_writeable(spa) && (state == SPA_LOAD_RECOVER ||
2614            spa->spa_load_max_txg == UINT64_MAX)) {
2615                dmu_tx_t *tx;
2616                int need_update = B_FALSE;

2618                ASSERT(state != SPA_LOAD_TRYIMPORT);

2620                /*
2621                 * Claim log blocks that haven't been committed yet.
2622                 * This must all happen in a single txg.
2623                 * Note: spa_claim_max_txg is updated by spa_claim_notify(),
2624                 * invoked from zil_claim_log_block()'s i/o done callback.
2625                 * Price of rollback is that we abandon the log.
2626                 */
2627                spa->spa_claiming = B_TRUE;

2629                tx = dmu_tx_create_assigned(spa_get_dsl(spa),
2630                    spa_first_txg(spa));
2631                (void) dmu_objset_find(spa_name(spa),
2632                    zil_claim, tx, DS_FIND_CHILDREN);
2633                dmu_tx_commit(tx);

2635                spa->spa_claiming = B_FALSE;

2637                spa_set_log_state(spa, SPA_LOG_GOOD);
2638                spa->spa_sync_on = B_TRUE;
2639                txg_sync_start(spa->spa_dsl_pool);

2641                /*
2642                 * Wait for all claims to sync.  We sync up to the highest
2643                 * claimed log block birth time so that claimed log blocks
2644                 * don't appear to be from the future.  spa_claim_max_txg
2645                 * will have been set for us by either zil_check_log_chain()
2646                 * (invoked from spa_check_logs()) or zil_claim() above.
2647                 */
2648                txg_wait_synced(spa->spa_dsl_pool, spa->spa_claim_max_txg);
```



```
2650                /*
2651                 * If the config cache is stale, or we have uninitialized
2652                 * metaslabs (see spa_vdev_add()), then update the config.
2653                 *
2654                 * If this is a verbatim import, trust the current
2655                 * in-core spa_config and update the disk labels.
2656                 */
2657                if (config_cache_txg != spa->spa_config_txg ||
2658                    state == SPA_LOAD_IMPORT ||
2659                    state == SPA_LOAD_RECOVER ||
2660                    (spa->spa_import_flags & ZFS_IMPORT_VERBATIM))
2661                        need_update = B_TRUE;

2663                for (int c = 0; c < rvd->vdev_children; c++)
2664                        if (rvd->vdev_child[c]->vdev_ms_array == 0)
2665                                need_update = B_TRUE;

2667                /*
2668                 * Update the config cache asychronously in case we're the
2669                 * root pool, in which case the config cache isn't writable yet.
2670                 */
2671                if (need_update)
2672                        spa_async_request(spa, SPA_ASYNC_CONFIG_UPDATE);

2674                /*
2675                 * Check all DTLs to see if anything needs resilvering.
2676                 */
2677                if (!dsl_scan_resilvering(spa->spa_dsl_pool) &&
2678                    vdev_resilver_needed(rvd, NULL, NULL))
2679                        spa_async_request(spa, SPA_ASYNC_RESILVER);

2681                /*
2682                 * Log the fact that we booted up (so that we can detect if
2683                 * we rebooted in the middle of an operation).
2684                 */
2685                spa_history_log_version(spa, "open");

2687                /*
2688                 * Delete any inconsistent datasets.
2689                 */
2690                (void) dmu_objset_find(spa_name(spa),
2691                    dsl_destroy_inconsistent, NULL, DS_FIND_CHILDREN);

2693                /*
2694                 * Clean up any stale temporary dataset userrefs.
2695                 */
2696                dsl_pool_clean_tmp_userrefs(spa->spa_dsl_pool);
2697        }

2699        return (0);
2700 }
_____unchanged_portion_omitted_

5896 /*
5897  * Set zpool properties.
5898  */
5899 static void
5900 spa_sync_props(void *arg, dmu_tx_t *tx)
5901 {
5902        nvlist_t *nvp = arg;
5903        spa_t *spa = dmu_tx_pool(tx)->dp_spa;
5904        objset_t *mos = spa->spa_meta_objset;
5905        nvpair_t *elem = NULL;

5907        mutex_enter(&spa->spa_props_lock);
```

```
5909         while ((elem = nvlist_next_nvpair(nvp, elem))) {
5910                 uint64_t intval;
5911                 char *strval, *fname;
5912                 zpool_prop_t prop;
5913                 const char *propname;
5914                 zprop_type_t proptype;
5915                 spa_feature_t fid;
5917                 zfeature_info_t *feature;

5917                 switch (prop = zpool_name_to_prop(nvpair_name(elem))) {
5918                 case ZPROP_INVAL:
5919                         /*
5920                          * We checked this earlier in spa_prop_validate().
5921                          */
5922                         ASSERT(zpool_prop_feature(nvpair_name(elem)));

5924                         fname = strchr(nvpair_name(elem), '@') + 1;
5925                         VERIFY0(zfeature_lookup_name(fname, &fid));
5927                         VERIFY0(zfeature_lookup_name(fname, &feature));

5927                         spa_feature_enable(spa, fid, tx);
5929                         spa_feature_enable(spa, feature, tx);
5928                         spa_history_log_internal(spa, "set", tx,
5929                             "%s=enabled", nvpair_name(elem));
5930                         break;

5932                 case ZPOOL_PROP_VERSION:
5933                         intval = fnvpair_value_uint64(elem);
5934                         /*
5935                          * The version is synced seperatly before other
5936                          * properties and should be correct by now.
5937                          */
5938                         ASSERT3U(spa_version(spa), >=, intval);
5939                         break;

5941                 case ZPOOL_PROP_ALTROOT:
5942                         /*
5943                          * 'altroot' is a non-persistent property. It should
5944                          * have been set temporarily at creation or import time.
5945                          */
5946                         ASSERT(spa->spa_root != NULL);
5947                         break;

5949                 case ZPOOL_PROP_READONLY:
5950                 case ZPOOL_PROP_CACHEFILE:
5951                         /*
5952                          * 'readonly' and 'cachefile' are also non-persisitent
5953                          * properties.
5954                          */
5955                         break;
5956                 case ZPOOL_PROP_COMMENT:
5957                         strval = fnvpair_value_string(elem);
5958                         if (spa->spa_comment != NULL)
5959                                 spa_strfree(spa->spa_comment);
5960                         spa->spa_comment = spa_strdup(strval);
5961                         /*
5962                          * We need to dirty the configuration on all the vdevs
5963                          * so that their labels get updated.  It's unnecessary
5964                          * to do this for pool creation since the vdev's
5965                          * configuratoin has already been dirtied.
5966                          */
5967                         if (tx->tx_txg != TXG_INITIAL)
5968                                 vdev_config_dirty(spa->spa_root_vdev);
5969                         spa_history_log_internal(spa, "set", tx,
5970                             "%s=%s", nvpair_name(elem), strval);
```



```
5971                         break;
5972                 default:
5973                         /*
5974                          * Set pool property values in the poolprops mos object.
5975                          */
5976                         if (spa->spa_pool_props_object == 0) {
5977                                 spa->spa_pool_props_object =
5978                                     zap_create_link(mos, DMU_OT_POOL_PROPS,
5979                                     DMU_POOL_DIRECTORY_OBJECT, DMU_POOL_PROPS,
5980                                     tx);
5981                         }

5983                         /* normalize the property name */
5984                         propname = zpool_prop_to_name(prop);
5985                         proptype = zpool_prop_get_type(prop);

5987                         if (nvpair_type(elem) == DATA_TYPE_STRING) {
5988                                 ASSERT(proptype == PROP_TYPE_STRING);
5989                                 strval = fnvpair_value_string(elem);
5990                                 VERIFY0(zap_update(mos,
5991                                     spa->spa_pool_props_object, propname,
5992                                     1, strlen(strval) + 1, strval, tx));
5993                                 spa_history_log_internal(spa, "set", tx,
5994                                     "%s=%s", nvpair_name(elem), strval);
5995                         } else if (nvpair_type(elem) == DATA_TYPE_UINT64) {
5996                                 intval = fnvpair_value_uint64(elem);

5998                                 if (proptype == PROP_TYPE_INDEX) {
5999                                         const char *unused;
6000                                         VERIFY0(zpool_prop_index_to_string(
6001                                             prop, intval, &unused));
6002                                 }
6003                                 VERIFY0(zap_update(mos,
6004                                     spa->spa_pool_props_object, propname,
6005                                     8, 1, &intval, tx));
6006                                 spa_history_log_internal(spa, "set", tx,
6007                                     "%s=%lld", nvpair_name(elem), intval);
6008                         } else {
6009                                 ASSERT(0); /* not allowed */
6010                         }

6012                         switch (prop) {
6013                         case ZPOOL_PROP_DELEGATION:
6014                                 spa->spa_delegation = intval;
6015                                 break;
6016                         case ZPOOL_PROP_BOOTFS:
6017                                 spa->spa_bootfs = intval;
6018                                 break;
6019                         case ZPOOL_PROP_FAILUREMODE:
6020                                 spa->spa_failmode = intval;
6021                                 break;
6022                         case ZPOOL_PROP_AUTOEXPAND:
6023                                 spa->spa_autoexpand = intval;
6024                                 if (tx->tx_txg != TXG_INITIAL)
6025                                         spa_async_request(spa,
6026                                             SPA_ASYNC_AUTOEXPAND);
6027                                 break;
6028                         case ZPOOL_PROP_DEDUPDITTO:
6029                                 spa->spa_dedup_ditto = intval;
6030                                 break;
6031                         default:
6032                                 break;
6033                         }
6034                 }
6036         }
```

```
6038         mutex_exit(&spa->spa_props_lock);
6039 }
```

_____***unchanged_portion_omitted_***

****
46279 Tue Oct 1 14:04:41 2013
new/usr/src/uts/common/fs/zfs/spa_misc.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****
_____***unchanged_portion_omitted_***

```
1137 /*
1138  * ==========================================================================
1139  * Miscellaneous functions
1140  * ==========================================================================
1141  */

1143 void
1144 spa_activate_mos_feature(spa_t *spa, const char *feature)
1145 {
1146         if (!nvlist_exists(spa->spa_label_features, feature)) {
1147                 fnvlist_add_boolean(spa->spa_label_features, feature);
1146         (void) nvlist_add_boolean(spa->spa_label_features, feature);
1148                 vdev_config_dirty(spa->spa_root_vdev);
1149         }
1150 }

1152 void
1153 spa_deactivate_mos_feature(spa_t *spa, const char *feature)
1154 {
1155         if (nvlist_remove_all(spa->spa_label_features, feature) == 0)
1153         (void) nvlist_remove_all(spa->spa_label_features, feature);
1156                 vdev_config_dirty(spa->spa_root_vdev);
1157 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   15742 Tue Oct  1 14:04:43 2013
new/usr/src/uts/common/fs/zfs/space_map.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
_____unchanged_portion_omitted_

 472 void
 473 space_map_truncate(space_map_t *sm, dmu_tx_t *tx)
 474 {
 475         objset_t *os = sm->sm_os;
 476         spa_t *spa = dmu_objset_spa(os);
 477         zfeature_info_t *space_map_histogram =
 478             &spa_feature_table[SPA_FEATURE_SPACEMAP_HISTOGRAM];
 477         dmu_object_info_t doi;
 478         int bonuslen;

 480         ASSERT(dsl_pool_sync_context(dmu_objset_pool(os)));
 481         ASSERT(dmu_tx_is_syncing(tx));

 483         VERIFY0(dmu_free_range(os, space_map_object(sm), 0, -1ULL, tx));
 484         dmu_object_info_from_db(sm->sm_dbuf, &doi);

 486         if (spa_feature_is_enabled(spa, SPA_FEATURE_SPACEMAP_HISTOGRAM)) {
 488         if (spa_feature_is_enabled(spa, space_map_histogram)) {
 487                 bonuslen = sizeof (space_map_phys_t);
 488                 ASSERT3U(bonuslen, <=, dmu_bonus_max());
 489         } else {
 490                 bonuslen = SPACE_MAP_SIZE_V0;
 491         }

 493         if (bonuslen != doi.doi_bonus_size ||
 494             doi.doi_data_block_size != SPACE_MAP_INITIAL_BLOCKSIZE) {
 495                 zfs_dbgmsg("txg %llu, spa %s, reallocating: "
 496                     "old bonus %u, old blocksz %u", dmu_tx_get_txg(tx),
 497                     spa_name(spa), doi.doi_bonus_size, doi.doi_data_block_size);
 498                 space_map_reallocate(sm, tx);
 499                 VERIFY3U(sm->sm_blksz, ==, SPACE_MAP_INITIAL_BLOCKSIZE);
 500         }

 502         dmu_buf_will_dirty(sm->sm_dbuf, tx);
 503         sm->sm_phys->smp_objsize = 0;
 504         sm->sm_phys->smp_alloc = 0;
 505 }
_____unchanged_portion_omitted_

 522 uint64_t
 523 space_map_alloc(objset_t *os, dmu_tx_t *tx)
 524 {
 525         spa_t *spa = dmu_objset_spa(os);
 528         zfeature_info_t *space_map_histogram =
 529             &spa_feature_table[SPA_FEATURE_SPACEMAP_HISTOGRAM];
 526         uint64_t object;
 527         int bonuslen;

 529         if (spa_feature_is_enabled(spa, SPA_FEATURE_SPACEMAP_HISTOGRAM)) {
 530                 spa_feature_incr(spa, SPA_FEATURE_SPACEMAP_HISTOGRAM, tx);
 533         if (spa_feature_is_enabled(spa, space_map_histogram)) {
 534                 spa_feature_incr(spa, space_map_histogram, tx);
 531                 bonuslen = sizeof (space_map_phys_t);
 532                 ASSERT3U(bonuslen, <=, dmu_bonus_max());
 533         } else {
```



```
 534                 bonuslen = SPACE_MAP_SIZE_V0;
 535         }

 537         object = dmu_object_alloc(os,
 538             DMU_OT_SPACE_MAP, SPACE_MAP_INITIAL_BLOCKSIZE,
 539             DMU_OT_SPACE_MAP_HEADER, bonuslen, tx);

 541         return (object);
 542 }

 544 void
 545 space_map_free(space_map_t *sm, dmu_tx_t *tx)
 546 {
 547         spa_t *spa;
 552         zfeature_info_t *space_map_histogram =
 553             &spa_feature_table[SPA_FEATURE_SPACEMAP_HISTOGRAM];

 549         if (sm == NULL)
 550                 return;

 552         spa = dmu_objset_spa(sm->sm_os);
 553         if (spa_feature_is_enabled(spa, SPA_FEATURE_SPACEMAP_HISTOGRAM)) {
 559         if (spa_feature_is_enabled(spa, space_map_histogram)) {
 554                 dmu_object_info_t doi;

 556                 dmu_object_info_from_db(sm->sm_dbuf, &doi);
 557                 if (doi.doi_bonus_size != SPACE_MAP_SIZE_V0) {
 558                         VERIFY(spa_feature_is_active(spa,
 559                             SPA_FEATURE_SPACEMAP_HISTOGRAM));
 560                         spa_feature_decr(spa,
 561                             SPA_FEATURE_SPACEMAP_HISTOGRAM, tx);
 564                         VERIFY(spa_feature_is_active(spa, space_map_histogram));
 565                         spa_feature_decr(spa, space_map_histogram, tx);
 562                 }
 563         }

 565         VERIFY3U(dmu_object_free(sm->sm_os, space_map_object(sm), tx), ==, 0);
 566         sm->sm_object = 0;
 567 }
_____unchanged_portion_omitted_
```

```
**********************************************************
    8514 Tue Oct  1 14:04:44 2013
new/usr/src/uts/common/fs/zfs/sys/dmu_impl.h
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 301 void dmu_object_zapify(objset_t *, uint64_t, dmu_object_type_t, dmu_tx_t *);
 302 void dmu_object_free_zapified(objset_t *, uint64_t, dmu_tx_t *);

 304 #ifdef  __cplusplus
 305 }
______unchanged_portion_omitted_
```

```
**********************************************************
   10682 Tue Oct  1 14:04:45 2013
new/usr/src/uts/common/fs/zfs/sys/dnode.h
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 146 typedef struct dnode {
 147         /*
 148          * Protects the structure of the dnode, including the number of levels
 149          * of indirection (dn_nlevels), dn_maxblkid, and dn_next_*
 150          */
 151         krwlock_t dn_struct_rwlock;

 153         /* Our link on dn_objset->os_dnodes list; protected by os_lock.  */
 154         list_node_t dn_link;

 156         /* immutable: */
 157         struct objset *dn_objset;
 158         uint64_t dn_object;
 159         struct dmu_buf_impl *dn_dbuf;
 160         struct dnode_handle *dn_handle;
 161         dnode_phys_t *dn_phys; /* pointer into dn->dn_dbuf->db.db_data */

 163         /*
 164          * Copies of stuff in dn_phys.  They're valid in the open
 165          * context (eg. even before the dnode is first synced).
 166          * Where necessary, these are protected by dn_struct_rwlock.
 167          */
 168         dmu_object_type_t dn_type;      /* object type */
 169         uint16_t dn_bonuslen;           /* bonus length */
 170         uint8_t dn_bonustype;           /* bonus type */
 171         uint8_t dn_nblkptr;             /* number of blkptrs (immutable) */
 172         uint8_t dn_checksum;            /* ZIO_CHECKSUM type */
 173         uint8_t dn_compress;            /* ZIO_COMPRESS type */
 174         uint8_t dn_nlevels;
 175         uint8_t dn_indblkshift;
 176         uint8_t dn_datablkshift;        /* zero if blksz not power of 2! */
 177         uint8_t dn_moved;               /* Has this dnode been moved? */
 178         uint16_t dn_datablkszsec;       /* in 512b sectors */
 179         uint32_t dn_datablksz;          /* in bytes */
 180         uint64_t dn_maxblkid;
 181         uint8_t dn_next_type[TXG_SIZE];
 182         uint8_t dn_next_nblkptr[TXG_SIZE];
 183         uint8_t dn_next_nlevels[TXG_SIZE];
 184         uint8_t dn_next_indblkshift[TXG_SIZE];
 185         uint8_t dn_next_bonustype[TXG_SIZE];
 186         uint8_t dn_rm_spillblk[TXG_SIZE];       /* for removing spill blk */
 187         uint16_t dn_next_bonuslen[TXG_SIZE];
 188         uint32_t dn_next_blksz[TXG_SIZE];       /* next block size in bytes */

 190         /* protected by dn_dbufs_mtx; declared here to fill 32-bit hole */
 191         uint32_t dn_dbufs_count;        /* count of dn_dbufs */
 192         /* There are no level-0 blocks of this blkid or higher in dn_dbufs */
 193         uint64_t dn_unlisted_l0_blkid;

 195         /* protected by os_lock: */
 196         list_node_t dn_dirty_link[TXG_SIZE];    /* next on dataset's dirty */

 198         /* protected by dn_mtx: */
 199         kmutex_t dn_mtx;
 200         list_t dn_dirty_records[TXG_SIZE];
```



```
 201         avl_tree_t dn_ranges[TXG_SIZE];
 202         uint64_t dn_allocated_txg;
 203         uint64_t dn_free_txg;
 204         uint64_t dn_assigned_txg;
 205         kcondvar_t dn_notxholds;
 206         enum dnode_dirtycontext dn_dirtyctx;
 207         uint8_t *dn_dirtyctx_firstset;          /* dbg: contents meaningless */

 209         /* protected by own devices */
 210         refcount_t dn_tx_holds;
 211         refcount_t dn_holds;

 213         kmutex_t dn_dbufs_mtx;
 214         list_t dn_dbufs;                /* descendent dbufs */

 216         /* protected by dn_struct_rwlock */
 217         struct dmu_buf_impl *dn_bonus;  /* bonus buffer dbuf */

 219         boolean_t dn_have_spill;        /* have spill or are spilling */

 221         /* parent IO for current sync write */
 222         zio_t *dn_zio;

 224         /* used in syncing context */
 225         uint64_t dn_oldused;    /* old phys used bytes */
 226         uint64_t dn_oldflags;   /* old phys dn_flags */
 227         uint64_t dn_olduid, dn_oldgid;
 228         uint64_t dn_newuid, dn_newgid;
 229         int dn_id_flags;

 231         /* holds prefetch structure */
 232         struct zfetch   dn_zfetch;
 233 } dnode_t;
______unchanged_portion_omitted_
```

```
**********************************************************
   10351 Tue Oct  1 14:04:46 2013
new/usr/src/uts/common/fs/zfs/sys/dsl_dataset.h
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright (c) 2005, 2010, Oracle and/or its affiliates. All rights reserved.
 * Copyright (c) 2013 by Delphix. All rights reserved.
 * Copyright (c) 2012, Joyent, Inc. All rights reserved.
 * Copyright (c) 2013 Steven Hartland. All rights reserved.
 */

#ifndef _SYS_DSL_DATASET_H
#define _SYS_DSL_DATASET_H

#include <sys/dmu.h>
#include <sys/spa.h>
#include <sys/txg.h>
#include <sys/zio.h>
#include <sys/bplist.h>
#include <sys/dsl_synctask.h>
#include <sys/zfs_context.h>
#include <sys/dsl_deadlist.h>
#include <sys/refcount.h>

#ifdef  __cplusplus
extern "C" {
#endif

struct dsl_dataset;
struct dsl_dir;
struct dsl_pool;

#define DS_FLAG_INCONSISTENT    (1ULL<<0)
#define DS_IS_INCONSISTENT(ds)  \
        ((ds)->ds_phys->ds_flags & DS_FLAG_INCONSISTENT)

/*
 * Do not allow this dataset to be promoted.
 * Note: nopromote can not yet be set, but we want support for it in this
 * on-disk version, so that we don't need to upgrade for it later.
 */
```



```
#define DS_FLAG_NOPROMOTE       (1ULL<<1)

/*
 * DS_FLAG_UNIQUE_ACCURATE is set if ds_unique_bytes has been correctly
 * calculated for head datasets (starting with SPA_VERSION_UNIQUE_ACCURATE,
 * refquota/refreservations).
 */
#define DS_FLAG_UNIQUE_ACCURATE (1ULL<<2)

/*
 * DS_FLAG_DEFER_DESTROY is set after 'zfs destroy -d' has been called
 * on a dataset. This allows the dataset to be destroyed using 'zfs release'.
 */
#define DS_FLAG_DEFER_DESTROY   (1ULL<<3)
#define DS_IS_DEFER_DESTROY(ds) \
        ((ds)->ds_phys->ds_flags & DS_FLAG_DEFER_DESTROY)

/*
 * DS_FIELD_* are strings that are used in the "extensified" dataset zap object.
 * They should be of the format <reverse-dns>:<field>.
 */

/*
 * DS_FLAG_CI_DATASET is set if the dataset contains a file system whose
 * name lookups should be performed case-insensitively.
 */
#define DS_FLAG_CI_DATASET      (1ULL<<16)

#define DS_CREATE_FLAG_NODIRTY  (1ULL<<24)

typedef struct dsl_dataset_phys {
        uint64_t ds_dir_obj;            /* DMU_OT_DSL_DIR */
        uint64_t ds_prev_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_prev_snap_txg;
        uint64_t ds_next_snap_obj;      /* DMU_OT_DSL_DATASET */
        uint64_t ds_snapnames_zapobj;   /* DMU_OT_DSL_DS_SNAP_MAP 0 for snaps */
        uint64_t ds_num_children;       /* clone/snap children; ==0 for head */
        uint64_t ds_creation_time;      /* seconds since 1970 */
        uint64_t ds_creation_txg;
        uint64_t ds_deadlist_obj;       /* DMU_OT_DEADLIST */
        /*
         * ds_referenced_bytes, ds_compressed_bytes, and ds_uncompressed_bytes
         * include all blocks referenced by this dataset, including those
         * shared with any other datasets.
         */
        uint64_t ds_referenced_bytes;
        uint64_t ds_compressed_bytes;
        uint64_t ds_uncompressed_bytes;
        uint64_t ds_unique_bytes;       /* only relevant to snapshots */
        /*
         * The ds_fsid_guid is a 56-bit ID that can change to avoid
         * collisions.  The ds_guid is a 64-bit ID that will never
         * change, so there is a small probability that it will collide.
         */
        uint64_t ds_fsid_guid;
        uint64_t ds_guid;
        uint64_t ds_flags;              /* DS_FLAG_* */
        blkptr_t ds_bp;
        uint64_t ds_next_clones_obj;    /* DMU_OT_DSL_CLONES */
        uint64_t ds_props_obj;          /* DMU_OT_DSL_PROPS for snaps */
        uint64_t ds_userrefs_obj;       /* DMU_OT_USERREFS */
        uint64_t ds_pad[5]; /* pad out to 320 bytes for good measure */
} dsl_dataset_phys_t;
```

*unchanged_portion_omitted_*

```
**********************************************************
   17686 Tue Oct  1 14:04:47 2013
new/usr/src/uts/common/fs/zfs/sys/zap.h
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 120 /*
 121  * Create a new zapobj with no attributes and return its object number.
 122  * MT_EXACT will cause the zap object to only support MT_EXACT lookups,
 123  * otherwise any matchtype can be used for lookups.
 124  *
 125  * normflags specifies what normalization will be done.  values are:
 126  * 0: no normalization (legacy on-disk format, supports MT_EXACT matching
 127  *     only)
 128  * U8_TEXTPREP_TOLOWER: case normalization will be performed.
 129  *     MT_FIRST/MT_BEST matching will find entries that match without
 130  *     regard to case (eg. looking for "foo" can find an entry "Foo").
 131  * Eventually, other flags will permit unicode normalization as well.
 132  */
 133 uint64_t zap_create(objset_t *ds, dmu_object_type_t ot,
 134     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
 135 uint64_t zap_create_norm(objset_t *ds, int normflags, dmu_object_type_t ot,
 136     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
 137 uint64_t zap_create_flags(objset_t *os, int normflags, zap_flags_t flags,
 138     dmu_object_type_t ot, int leaf_blockshift, int indirect_blockshift,
 139     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
 140 uint64_t zap_create_link(objset_t *os, dmu_object_type_t ot,
 141     uint64_t parent_obj, const char *name, dmu_tx_t *tx);

 143 /*
 144  * Initialize an already-allocated object.
 145  */
 146 void mzap_create_impl(objset_t *os, uint64_t obj, int normflags,
 147     zap_flags_t flags, dmu_tx_t *tx);

 149 /*
 150  * Create a new zapobj with no attributes from the given (unallocated)
 151  * object number.
 152  */
 153 int zap_create_claim(objset_t *ds, uint64_t obj, dmu_object_type_t ot,
 154     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);
 155 int zap_create_claim_norm(objset_t *ds, uint64_t obj,
 156     int normflags, dmu_object_type_t ot,
 157     dmu_object_type_t bonustype, int bonuslen, dmu_tx_t *tx);

 159 /*
 160  * The zapobj passed in must be a valid ZAP object for all of the
 161  * following routines.
 162  */

 164 /*
 165  * Destroy this zapobj and all its attributes.
 166  *
 167  * Frees the object number using dmu_object_free.
 168  */
 169 int zap_destroy(objset_t *ds, uint64_t zapobj, dmu_tx_t *tx);

 171 /*
 172  * Manipulate attributes.
 173  *
 174  * 'integer_size' is in bytes, and must be 1, 2, 4, or 8.
```



```
 175  */

 177 /*
 178  * Retrieve the contents of the attribute with the given name.
 179  *
 180  * If the requested attribute does not exist, the call will fail and
 181  * return ENOENT.
 182  *
 183  * If 'integer_size' is smaller than the attribute's integer size, the
 184  * call will fail and return EINVAL.
 185  *
 186  * If 'integer_size' is equal to or larger than the attribute's integer
 187  * size, the call will succeed and return 0.
 188  *
 189  * When converting to a larger integer size, the integers will be treated as
 190  * unsigned (ie. no sign-extension will be performed).
 191  *
 192  * 'num_integers' is the length (in integers) of 'buf'.
 193  *
 194  * If the attribute is longer than the buffer, as many integers as will
 195  * fit will be transferred to 'buf'.  If the entire attribute was not
 196  * transferred, the call will return EOVERFLOW.
 197  */
 198 int zap_lookup(objset_t *ds, uint64_t zapobj, const char *name,
 199     uint64_t integer_size, uint64_t num_integers, void *buf);

 201 /*
 202  * If rn_len is nonzero, realname will be set to the name of the found
 203  * entry (which may be different from the requested name if matchtype is
 204  * not MT_EXACT).
 205  *
 206  * If normalization_conflictp is not NULL, it will be set if there is
 207  * another name with the same case/unicode normalized form.
 208  */
 209 int zap_lookup_norm(objset_t *ds, uint64_t zapobj, const char *name,
 210     uint64_t integer_size, uint64_t num_integers, void *buf,
 211     matchtype_t mt, char *realname, int rn_len,
 212     boolean_t *normalization_conflictp);
 213 int zap_lookup_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
 214     int key_numints, uint64_t integer_size, uint64_t num_integers, void *buf);
 215 int zap_contains(objset_t *ds, uint64_t zapobj, const char *name);
 216 int zap_prefetch_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
 217     int key_numints);

 219 int zap_count_write(objset_t *os, uint64_t zapobj, const char *name,
 220     int add, uint64_t *towrite, uint64_t *tooverwrite);

 222 /*
 223  * Create an attribute with the given name and value.
 224  *
 225  * If an attribute with the given name already exists, the call will
 226  * fail and return EEXIST.
 227  */
 228 int zap_add(objset_t *ds, uint64_t zapobj, const char *key,
 229     int integer_size, uint64_t num_integers,
 230     const void *val, dmu_tx_t *tx);
 231 int zap_add_uint64(objset_t *ds, uint64_t zapobj, const uint64_t *key,
 232     int key_numints, int integer_size, uint64_t num_integers,
 233     const void *val, dmu_tx_t *tx);

 235 /*
 236  * Set the attribute with the given name to the given value.  If an
 237  * attribute with the given name does not exist, it will be created.  If
 238  * an attribute with the given name already exists, the previous value
 239  * will be overwritten.  The integer_size may be different from the
 240  * existing attribute's integer size, in which case the attribute's
```

```
 241  * integer size will be updated to the new value.
 242  */
 243 int zap_update(objset_t *ds, uint64_t zapobj, const char *name,
 244     int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx);
 245 int zap_update_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
 246     int key_numints,
 247     int integer_size, uint64_t num_integers, const void *val, dmu_tx_t *tx);

 249 /*
 250  * Get the length (in integers) and the integer size of the specified
 251  * attribute.
 252  *
 253  * If the requested attribute does not exist, the call will fail and
 254  * return ENOENT.
 255  */
 256 int zap_length(objset_t *ds, uint64_t zapobj, const char *name,
 257     uint64_t *integer_size, uint64_t *num_integers);
 258 int zap_length_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
 259     int key_numints, uint64_t *integer_size, uint64_t *num_integers);

 261 /*
 262  * Remove the specified attribute.
 263  *
 264  * If the specified attribute does not exist, the call will fail and
 265  * return ENOENT.
 266  */
 267 int zap_remove(objset_t *ds, uint64_t zapobj, const char *name, dmu_tx_t *tx);
 268 int zap_remove_norm(objset_t *ds, uint64_t zapobj, const char *name,
 269     matchtype_t mt, dmu_tx_t *tx);
 270 int zap_remove_uint64(objset_t *os, uint64_t zapobj, const uint64_t *key,
 271     int key_numints, dmu_tx_t *tx);

 273 /*
 274  * Returns (in *count) the number of attributes in the specified zap
 275  * object.
 276  */
 277 int zap_count(objset_t *ds, uint64_t zapobj, uint64_t *count);

 279 /*
 280  * Returns (in name) the name of the entry whose (value & mask)
 281  * (za_first_integer) is value, or ENOENT if not found.  The string
 282  * pointed to by name must be at least 256 bytes long.  If mask==0, the
 283  * match must be exact (ie, same as mask=-1ULL).
 284  */
 285 int zap_value_search(objset_t *os, uint64_t zapobj,
 286     uint64_t value, uint64_t mask, char *name);

 288 /*
 289  * Transfer all the entries from fromobj into intoobj.  Only works on
 290  * int_size=8 num_integers=1 values.  Fails if there are any duplicated
 291  * entries.
 292  */
 293 int zap_join(objset_t *os, uint64_t fromobj, uint64_t intoobj, dmu_tx_t *tx);

 295 /* Same as zap_join, but set the values to 'value'. */
 296 int zap_join_key(objset_t *os, uint64_t fromobj, uint64_t intoobj,
 297     uint64_t value, dmu_tx_t *tx);

 299 /* Same as zap_join, but add together any duplicated entries. */
 300 int zap_join_increment(objset_t *os, uint64_t fromobj, uint64_t intoobj,
 301     dmu_tx_t *tx);

 303 /*
 304  * Manipulate entries where the name + value are the "same" (the name is
 305  * a stringified version of the value).
 306  */
```



```
 307 int zap_add_int(objset_t *os, uint64_t obj, uint64_t value, dmu_tx_t *tx);
 308 int zap_remove_int(objset_t *os, uint64_t obj, uint64_t value, dmu_tx_t *tx);
 309 int zap_lookup_int(objset_t *os, uint64_t obj, uint64_t value);
 310 int zap_increment_int(objset_t *os, uint64_t obj, uint64_t key, int64_t delta,
 311     dmu_tx_t *tx);

 313 /* Here the key is an int and the value is a different int. */
 314 int zap_add_int_key(objset_t *os, uint64_t obj,
 315     uint64_t key, uint64_t value, dmu_tx_t *tx);
 316 int zap_update_int_key(objset_t *os, uint64_t obj,
 317     uint64_t key, uint64_t value, dmu_tx_t *tx);
 318 int zap_lookup_int_key(objset_t *os, uint64_t obj,
 319     uint64_t key, uint64_t *valuep);

 321 int zap_increment(objset_t *os, uint64_t obj, const char *name, int64_t delta,
 322     dmu_tx_t *tx);

 324 struct zap;
 325 struct zap_leaf;
 326 typedef struct zap_cursor {
 327         /* This structure is opaque! */
 328         objset_t *zc_objset;
 329         struct zap *zc_zap;
 330         struct zap_leaf *zc_leaf;
 331         uint64_t zc_zapobj;
 332         uint64_t zc_serialized;
 333         uint64_t zc_hash;
 334         uint32_t zc_cd;
 335 } zap_cursor_t;
```

_____***unchanged_portion_omitted_***

```
**********************************************************
    2121 Tue Oct  1 14:04:47 2013
new/usr/src/uts/common/fs/zfs/sys/zfeature.h
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#ifndef _SYS_ZFEATURE_H
#define _SYS_ZFEATURE_H

#include <sys/nvpair.h>
#include <sys/txg.h>
#include "zfeature_common.h"

#ifdef  __cplusplus
extern "C" {
#endif

struct spa;
struct dmu_tx;
struct objset;

extern boolean_t feature_is_supported(struct objset *os, uint64_t obj,
    uint64_t desc_obj, nvlist_t *unsup_feat, nvlist_t *enabled_feat);

extern void spa_feature_create_zap_objects(struct spa *, struct dmu_tx *);
extern void spa_feature_enable(struct spa *, spa_feature_t,
extern void spa_feature_enable(struct spa *, zfeature_info_t *,
    struct dmu_tx *);
extern void spa_feature_incr(struct spa *, spa_feature_t, struct dmu_tx *);
extern void spa_feature_decr(struct spa *, spa_feature_t, struct dmu_tx *);
extern boolean_t spa_feature_is_enabled(struct spa *, spa_feature_t);
extern boolean_t spa_feature_is_active(struct spa *, spa_feature_t);
extern uint64_t spa_feature_refcount(spa_t *, spa_feature_t, uint64_t);
extern boolean_t spa_features_check(spa_t *, boolean_t, nvlist_t *, nvlist_t *);
extern void spa_feature_incr(struct spa *, zfeature_info_t *, struct dmu_tx *);
extern void spa_feature_decr(struct spa *, zfeature_info_t *, struct dmu_tx *);
extern boolean_t spa_feature_is_enabled(struct spa *, zfeature_info_t *);
extern boolean_t spa_feature_is_active(struct spa *, zfeature_info_t *);
```



```
extern int spa_feature_get_refcount(struct spa *, zfeature_info_t *);

/*
 * These functions are only exported for zhack and zdb; normal callers should
 * use the above interfaces.
 */
extern int feature_get_refcount(struct spa *, zfeature_info_t *, uint64_t *);
extern void feature_enable_sync(struct spa *, zfeature_info_t *,
    struct dmu_tx *);
extern void feature_sync(struct spa *, zfeature_info_t *, uint64_t,
    struct dmu_tx *);

#ifdef  __cplusplus
}
_____unchanged_portion_omitted_
```

****
34195 Tue Oct 1 14:04:48 2013
new/usr/src/uts/common/fs/zfs/zap_micro.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
****
_____***unchanged_portion_omitted_***

```
 575 void
 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                 VERIFY3U(0, ==, mzap_upgrade(&zap, tx, flags));
 605                 zap_unlockdir(zap);
 606         }
 607 }
```

_____***unchanged_portion_omitted_***

```
 862 int
 863 zap_contains(objset_t *os, uint64_t zapobj, const char *name)
 864 {
 865         int err = zap_lookup_norm(os, zapobj, name, 0,
 866             0, NULL, MT_EXACT, NULL, 0, NULL);
 865         int err = (zap_lookup_norm(os, zapobj, name, 0,
 866             0, NULL, MT_EXACT, NULL, 0, NULL));
 867         if (err == EOVERFLOW || err == EINVAL)
 868                 err = 0; /* found, but skipped reading the value */
 869         return (err);
 870 }
```

_____***unchanged_portion_omitted_***

```
**********************************************************
   14027 Tue Oct  1 14:04:49 2013
new/usr/src/uts/common/fs/zfs/zfeature.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
```

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

/*
 * Copyright (c) 2013 by Delphix. All rights reserved.
 */

#include <sys/zfs_context.h>
#include <sys/zfeature.h>
#include <sys/dmu.h>
#include <sys/nvpair.h>
#include <sys/zap.h>
#include <sys/dmu_tx.h>
#include "zfeature_common.h"
#include <sys/spa_impl.h>

/*
 * ZFS Feature Flags
 * -----------------
 *
 * ZFS feature flags are used to provide fine-grained versioning to the ZFS
 * on-disk format. Once enabled on a pool feature flags replace the old
 * spa_version() number.
 *
 * Each new on-disk format change will be given a uniquely identifying string
 * guid rather than a version number. This avoids the problem of different
 * organizations creating new on-disk formats with the same version number. To
 * keep feature guids unique they should consist of the reverse dns name of the
 * organization which implemented the feature and a short name for the feature,
 * separated by a colon (e.g. com.delphix:async_destroy).
 *
 * Reference Counts
 * ----------------
 *
 * Within each pool features can be in one of three states: disabled, enabled,
 * or active. These states are differentiated by a reference count stored on
 * disk for each feature:
 *
 *   1) If there is no reference count stored on disk the feature is disabled.
```



```
 *   2) If the reference count is 0 a system administrator has enabled the
 *      feature, but the feature has not been used yet, so no on-disk
 *      format changes have been made.
 *   3) If the reference count is greater than 0 the feature is active.
 *      The format changes required by the feature are currently on disk.
 *      Note that if the feature's format changes are reversed the feature
 *      may choose to set its reference count back to 0.
 *
 * Feature flags makes no differentiation between non-zero reference counts
 * for an active feature (e.g. a reference count of 1 means the same thing as a
 * reference count of 27834721), but feature implementations may choose to use
 * the reference count to store meaningful information. For example, a new RAID
 * implementation might set the reference count to the number of vdevs using
 * it. If all those disks are removed from the pool the feature goes back to
 * having a reference count of 0.
 *
 * It is the responsibility of the individual features to maintain a non-zero
 * reference count as long as the feature's format changes are present on disk.
 *
 * Dependencies
 * ------------
 *
 * Each feature may depend on other features. The only effect of this
 * relationship is that when a feature is enabled all of its dependencies are
 * automatically enabled as well. Any future work to support disabling of
 * features would need to ensure that features cannot be disabled if other
 * enabled features depend on them.
 *
 * On-disk Format
 * --------------
 *
 * When feature flags are enabled spa_version() is set to SPA_VERSION_FEATURES
 * (5000). In order for this to work the pool is automatically upgraded to
 * SPA_VERSION_BEFORE_FEATURES (28) first, so all pre-feature flags on disk
 * format changes will be in use.
 *
 * Information about features is stored in 3 ZAP objects in the pool's MOS.
 * These objects are linked to by the following names in the pool directory
 * object:
 *
 * 1) features_for_read: feature guid -> reference count
 *    Features needed to open the pool for reading.
 * 2) features_for_write: feature guid -> reference count
 *    Features needed to open the pool for writing.
 * 3) feature_descriptions: feature guid -> descriptive string
 *    A human readable string.
 *
 * All enabled features appear in either features_for_read or
 * features_for_write, but not both.
 *
 * To open a pool in read-only mode only the features listed in
 * features_for_read need to be supported.
 *
 * To open the pool in read-write mode features in both features_for_read and
 * features_for_write need to be supported.
 *
 * Some features may be required to read the ZAP objects containing feature
 * information. To allow software to check for compatibility with these features
 * before the pool is opened their names must be stored in the label in a
 * new "features_for_read" entry (note that features that are only required
 * to write to a pool never need to be stored in the label since the
 * features_for_write ZAP object can be read before the pool is written to).
 * To save space in the label features must be explicitly marked as needing to
 * be written to the label. Also, reference counts are not stored in the label,
 * instead any feature whose reference count drops to 0 is removed from the
 * label.
```

```
 *
 * Adding New Features
 * -------------------
 *
 * Features must be registered in zpool_feature_init() function in
 * zfeature_common.c using the zfeature_register() function. This function
 * has arguments to specify if the feature should be stored in the
 * features_for_read or features_for_write ZAP object and if it needs to be
 * written to the label when active.
 *
 * Once a feature is registered it will appear as a "feature@<feature name>"
 * property which can be set by an administrator. Feature implementors should
 * use the spa_feature_is_enabled() and spa_feature_is_active() functions to
 * query the state of a feature and the spa_feature_incr() and
 * spa_feature_decr() functions to change an enabled feature's reference count.
 * Reference counts may only be updated in the syncing context.
 *
 * Features may not perform enable-time initialization. Instead, any such
 * initialization should occur when the feature is first used. This design
 * enforces that on-disk changes be made only when features are used. Code
 * should only check if a feature is enabled using spa_feature_is_enabled(),
 * not by relying on any feature specific metadata existing. If a feature is
 * enabled, but the feature's metadata is not on disk yet then it should be
 * created as needed.
 *
 * As an example, consider the com.delphix:async_destroy feature. This feature
 * relies on the existence of a bptree in the MOS that store blocks for
 * asynchronous freeing. This bptree is not created when async_destroy is
 * enabled. Instead, when a dataset is destroyed spa_feature_is_enabled() is
 * called to check if async_destroy is enabled. If it is and the bptree object
 * does not exist yet, the bptree object is created as part of the dataset
 * destroy and async_destroy's reference count is incremented to indicate it
 * has made an on-disk format change. Later, after the destroyed dataset's
 * blocks have all been asynchronously freed there is no longer any use for the
 * bptree object, so it is destroyed and async_destroy's reference count is
 * decremented back to 0 to indicate that it has undone its on-disk format
 * changes.
 */

typedef enum {
        FEATURE_ACTION_ENABLE,
        FEATURE_ACTION_INCR,
        FEATURE_ACTION_DECR,
} feature_action_t;

/*
 * Checks that the active features in the pool are supported by
 * Checks that the features active in the specified object are supported by
 * this software.  Adds each unsupported feature (name -> description) to
 * the supplied nvlist.
 */
boolean_t
spa_features_check(spa_t *spa, boolean_t for_write,
feature_is_supported(objset_t *os, uint64_t obj, uint64_t desc_obj,
    nvlist_t *unsup_feat, nvlist_t *enabled_feat)
{
        objset_t *os = spa->spa_meta_objset;
        boolean_t supported;
        zap_cursor_t zc;
        zap_attribute_t za;
        uint64_t obj = for_write ?
            spa->spa_feat_for_write_obj : spa->spa_feat_for_read_obj;

        supported = B_TRUE;
        for (zap_cursor_init(&zc, os, obj);
            zap_cursor_retrieve(&zc, &za) == 0;
```



```
            zap_cursor_advance(&zc)) {
                ASSERT(za.za_integer_length == sizeof (uint64_t) &&
                    za.za_num_integers == 1);

                if (NULL != enabled_feat) {
                        fnvlist_add_uint64(enabled_feat, za.za_name,
                            za.za_first_integer);
                }

                if (za.za_first_integer != 0 &&
                    !zfeature_is_supported(za.za_name)) {
                        supported = B_FALSE;

                        if (NULL != unsup_feat) {
                                char *desc = "";
                                char buf[MAXPATHLEN];

                                if (zap_lookup(os, spa->spa_feat_desc_obj,
                                    za.za_name, 1, sizeof (buf), buf) == 0)
                                if (zap_lookup(os, desc_obj, za.za_name,
                                    1, sizeof (buf), buf) == 0)
                                        desc = buf;

                                VERIFY(nvlist_add_string(unsup_feat, za.za_name,
                                    desc) == 0);
                        }
                }
        }
        zap_cursor_fini(&zc);

        return (supported);
}

/*
 * Note: well-designed features will not need to use this; they should
 * use spa_feature_is_enabled() and spa_feature_is_active() instead.
 * However, this is non-static for zdb and zhack.
 */
int
feature_get_refcount(spa_t *spa, zfeature_info_t *feature, uint64_t *res)
static int
feature_get_refcount(objset_t *os, uint64_t read_obj, uint64_t write_obj,
    zfeature_info_t *feature, uint64_t *res)
{
        int err;
        uint64_t refcount;
        uint64_t zapobj = feature->fi_can_readonly ?
            spa->spa_feat_for_write_obj : spa->spa_feat_for_read_obj;
        uint64_t zapobj = feature->fi_can_readonly ? write_obj : read_obj;

        /*
         * If the pool is currently being created, the feature objects may not
         * have been allocated yet.  Act as though all features are disabled.
         */
        if (zapobj == 0)
                return (SET_ERROR(ENOTSUP));

        err = zap_lookup(spa->spa_meta_objset, zapobj,
            feature->fi_guid, sizeof (uint64_t), 1, &refcount);
        err = zap_lookup(os, zapobj, feature->fi_guid, sizeof (uint64_t), 1,
            &refcount);
        if (err != 0) {
                if (err == ENOENT)
                        return (SET_ERROR(ENOTSUP));
                else
                        return (err);
```

```
        }
        *res = refcount;
        return (0);
}

/*
 * This function is non-static for zhack; it should otherwise not be used
 * outside this file.
 */
void
feature_sync(spa_t *spa, zfeature_info_t *feature, uint64_t refcount,
static int
feature_do_action(objset_t *os, uint64_t read_obj, uint64_t write_obj,
    uint64_t desc_obj, zfeature_info_t *feature, feature_action_t action,
    dmu_tx_t *tx)
{
        uint64_t zapobj = feature->fi_can_readonly ?
            spa->spa_feat_for_write_obj : spa->spa_feat_for_read_obj;
        int error;
        uint64_t refcount;
        uint64_t zapobj = feature->fi_can_readonly ? write_obj : read_obj;

        VERIFY0(zap_update(spa->spa_meta_objset, zapobj, feature->fi_guid,
            sizeof (uint64_t), 1, &refcount, tx));

        if (refcount == 0)
                spa_deactivate_mos_feature(spa, feature->fi_guid);
        else if (feature->fi_mos)
                spa_activate_mos_feature(spa, feature->fi_guid);
}

/*
 * This function is non-static for zhack; it should otherwise not be used
 * outside this file.
 */
void
feature_enable_sync(spa_t *spa, zfeature_info_t *feature, dmu_tx_t *tx)
{
        uint64_t zapobj = feature->fi_can_readonly ?
            spa->spa_feat_for_write_obj : spa->spa_feat_for_read_obj;

        ASSERT(0 != zapobj);
        ASSERT(zfeature_is_valid_guid(feature->fi_guid));
        ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);

        error = zap_lookup(os, zapobj, feature->fi_guid,
            sizeof (uint64_t), 1, &refcount);

        /*
         * If the feature is already enabled, ignore the request.
         * If we can't ascertain the status of the specified feature, an I/O
         * error occurred.
         */
        if (zap_contains(spa->spa_meta_objset, zapobj, feature->fi_guid) == 0)
                return;
        if (error != 0 && error != ENOENT)
                return (error);

        for (int i = 0; feature->fi_depends[i] != SPA_FEATURE_NONE; i++)
                spa_feature_enable(spa, feature->fi_depends[i], tx);

        VERIFY0(zap_update(spa->spa_meta_objset, spa->spa_feat_desc_obj,
            feature->fi_guid, 1, strlen(feature->fi_desc) + 1,
            feature->fi_desc, tx));
        feature_sync(spa, feature, 0, tx);
}
```



```
static void
feature_do_action(spa_t *spa, spa_feature_t fid, feature_action_t action,
    dmu_tx_t *tx)
{
        uint64_t refcount;
        zfeature_info_t *feature = &spa_feature_table[fid];
        uint64_t zapobj = feature->fi_can_readonly ?
            spa->spa_feat_for_write_obj : spa->spa_feat_for_read_obj;

        ASSERT3U(fid, <, SPA_FEATURES);
        ASSERT(0 != zapobj);
        ASSERT(zfeature_is_valid_guid(feature->fi_guid));

        ASSERT(dmu_tx_is_syncing(tx));
        ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);

        VERIFY0(zap_lookup(spa->spa_meta_objset, zapobj, feature->fi_guid,
            sizeof (uint64_t), 1, &refcount));

        switch (action) {
        case FEATURE_ACTION_ENABLE:
                /*
                 * If the feature is already enabled, ignore the request.
                 */
                if (error == 0)
                        return (0);
                refcount = 0;
                break;
        case FEATURE_ACTION_INCR:
                VERIFY3U(refcount, !=, UINT64_MAX);
                if (error == ENOENT)
                        return (SET_ERROR(ENOTSUP));
                if (refcount == UINT64_MAX)
                        return (SET_ERROR(EOVERFLOW));
                refcount++;
                break;
        case FEATURE_ACTION_DECR:
                VERIFY3U(refcount, !=, 0);
                if (error == ENOENT)
                        return (SET_ERROR(ENOTSUP));
                if (refcount == 0)
                        return (SET_ERROR(EOVERFLOW));
                refcount--;
                break;
        default:
                ASSERT(0);
                break;
        }

        feature_sync(spa, feature, refcount, tx);
        if (action == FEATURE_ACTION_ENABLE) {
                int i;

                for (i = 0; feature->fi_depends[i] != NULL; i++) {
                        zfeature_info_t *dep = feature->fi_depends[i];

                        error = feature_do_action(os, read_obj, write_obj,
                            desc_obj, dep, FEATURE_ACTION_ENABLE, tx);
                        if (error != 0)
                                return (error);
                }
        }

        error = zap_update(os, zapobj, feature->fi_guid,
            sizeof (uint64_t), 1, &refcount, tx);
```

```
308         if (error != 0)
309                 return (error);
311         if (action == FEATURE_ACTION_ENABLE) {
312                 error = zap_update(os, desc_obj,
313                     feature->fi_guid, 1, strlen(feature->fi_desc) + 1,
314                     feature->fi_desc, tx);
315                 if (error != 0)
316                         return (error);
317         }
319         if (action == FEATURE_ACTION_INCR && refcount == 1 && feature->fi_mos) {
320                 spa_activate_mos_feature(dmu_objset_spa(os), feature->fi_guid);
321         }
323         if (action == FEATURE_ACTION_DECR && refcount == 0) {
324                 spa_deactivate_mos_feature(dmu_objset_spa(os),
325                     feature->fi_guid);
326         }
328         return (0);
333 }
```

_____*unchanged_portion_omitted_*

```
356 /*
357  * Enable any required dependencies, then enable the requested feature.
358  */
359 void
360 spa_feature_enable(spa_t *spa, spa_feature_t fid, dmu_tx_t *tx)
356 spa_feature_enable(spa_t *spa, zfeature_info_t *feature, dmu_tx_t *tx)
361 {
362         ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);
363         ASSERT3U(fid, <, SPA_FEATURES);
364         feature_enable_sync(spa, &spa_feature_table[fid], tx);
359         VERIFY3U(0, ==, feature_do_action(spa->spa_meta_objset,
360             spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
361             spa->spa_feat_desc_obj, feature, FEATURE_ACTION_ENABLE, tx));
365 }
367 void
368 spa_feature_incr(spa_t *spa, spa_feature_t fid, dmu_tx_t *tx)
365 spa_feature_incr(spa_t *spa, zfeature_info_t *feature, dmu_tx_t *tx)
369 {
370         feature_do_action(spa, fid, FEATURE_ACTION_INCR, tx);
367         ASSERT(dmu_tx_is_syncing(tx));
368         ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);
369         VERIFY3U(0, ==, feature_do_action(spa->spa_meta_objset,
370             spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
371             spa->spa_feat_desc_obj, feature, FEATURE_ACTION_INCR, tx));
371 }
373 void
374 spa_feature_decr(spa_t *spa, spa_feature_t fid, dmu_tx_t *tx)
375 spa_feature_decr(spa_t *spa, zfeature_info_t *feature, dmu_tx_t *tx)
375 {
376         feature_do_action(spa, fid, FEATURE_ACTION_DECR, tx);
377         ASSERT(dmu_tx_is_syncing(tx));
378         ASSERT3U(spa_version(spa), >=, SPA_VERSION_FEATURES);
379         VERIFY3U(0, ==, feature_do_action(spa->spa_meta_objset,
380             spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
381             spa->spa_feat_desc_obj, feature, FEATURE_ACTION_DECR, tx));
377 }
384 /*
385  * This interface is for debugging only. Normal consumers should use
386  * spa_feature_is_enabled/spa_feature_is_active.
```



```
387  */
388 int
389 spa_feature_get_refcount(spa_t *spa, zfeature_info_t *feature)
390 {
391         int err;
392         uint64_t refcount;
394         if (spa_version(spa) < SPA_VERSION_FEATURES)
395                 return (B_FALSE);
397         err = feature_get_refcount(spa->spa_meta_objset,
398             spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
399             feature, &refcount);
400         ASSERT(err == 0 || err == ENOTSUP);
401         return (err == 0 ? refcount : 0);
402 }
379 boolean_t
380 spa_feature_is_enabled(spa_t *spa, spa_feature_t fid)
405 spa_feature_is_enabled(spa_t *spa, zfeature_info_t *feature)
381 {
382         int err;
383         uint64_t refcount;
385         ASSERT3U(fid, <, SPA_FEATURES);
386         if (spa_version(spa) < SPA_VERSION_FEATURES)
387                 return (B_FALSE);
389         err = feature_get_refcount(spa, &spa_feature_table[fid], &refcount);
413         err = feature_get_refcount(spa->spa_meta_objset,
414             spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
415             feature, &refcount);
390         ASSERT(err == 0 || err == ENOTSUP);
391         return (err == 0);
392 }
394 boolean_t
395 spa_feature_is_active(spa_t *spa, spa_feature_t fid)
421 spa_feature_is_active(spa_t *spa, zfeature_info_t *feature)
396 {
397         int err;
398         uint64_t refcount;
400         ASSERT3U(fid, <, SPA_FEATURES);
401         if (spa_version(spa) < SPA_VERSION_FEATURES)
402                 return (B_FALSE);
404         err = feature_get_refcount(spa, &spa_feature_table[fid], &refcount);
429         err = feature_get_refcount(spa->spa_meta_objset,
430             spa->spa_feat_for_read_obj, spa->spa_feat_for_write_obj,
431             feature, &refcount);
405         ASSERT(err == 0 || err == ENOTSUP);
406         return (err == 0 && refcount > 0);
407 }
```

_____*unchanged_portion_omitted_*

```
**********************************************************
  144359 Tue Oct  1 14:04:49 2013
new/usr/src/uts/common/fs/zfs/zfs_ioctl.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

 235 static int zfs_ioc_userspace_upgrade(zfs_cmd_t *zc);
 236 static int zfs_check_settable(const char *name, nvpair_t *property,
 237     cred_t *cr);
 238 static int zfs_check_clearable(char *dataset, nvlist_t *props,
 239     nvlist_t **errors);
 240 static int zfs_fill_zplprops_root(uint64_t, nvlist_t *, nvlist_t *,
 241     boolean_t *);
 242 int zfs_set_prop_nvlist(const char *, zprop_source_t, nvlist_t *, nvlist_t *);
 243 static int get_nvlist(uint64_t nvl, uint64_t size, int iflag, nvlist_t **nvp);

 245 static int zfs_prop_activate_feature(spa_t *spa, spa_feature_t feature);
 245 static int zfs_prop_activate_feature(spa_t *spa, zfeature_info_t *feature);

 247 /* _NOTE(PRINTFLIKE(4)) - this is printf-like, but lint is too whiney */
 248 void
 249 __dprintf(const char *file, const char *func, int line, const char *fmt, ...)
 250 {
 251         const char *newfile;
 252         char buf[512];
 253         va_list adx;

 255         /*
 256          * Get rid of annoying "../common/" prefix to filename.
 257          */
 258         newfile = strrchr(file, '/');
 259         if (newfile != NULL) {
 260                 newfile = newfile + 1; /* Get rid of leading / */
 261         } else {
 262                 newfile = file;
 263         }

 265         va_start(adx, fmt);
 266         (void) vsnprintf(buf, sizeof (buf), fmt, adx);
 267         va_end(adx);

 269         /*
 270          * To get this data, use the zfs-dprintf probe as so:
 271          * dtrace -q -n 'zfs-dprintf \
 272          *      /stringof(arg0) == "dbuf.c"/ \
 273          *      {printf("%s: %s", stringof(arg1), stringof(arg3))}'
 274          * arg0 = file name
 275          * arg1 = function name
 276          * arg2 = line number
 277          * arg3 = message
 278          */
 279         DTRACE_PROBE4(zfs__dprintf,
 280             char *, newfile, char *, func, int, line, char *, buf);
 281 }
______unchanged_portion_omitted_

2311 /*
2312  * If the named property is one that has a special function to set its value,
2313  * return 0 on success and a positive error code on failure; otherwise if it is
2314  * not one of the special properties handled by this function, return -1.
2315  *
```



```
2316  * XXX: It would be better for callers of the property interface if we handled
2317  * these special cases in dsl_prop.c (in the dsl layer).
2318  */
2319 static int
2320 zfs_prop_set_special(const char *dsname, zprop_source_t source,
2321     nvpair_t *pair)
2322 {
2323         const char *propname = nvpair_name(pair);
2324         zfs_prop_t prop = zfs_name_to_prop(propname);
2325         uint64_t intval;
2326         int err;

2328         if (prop == ZPROP_INVAL) {
2329                 if (zfs_prop_userquota(propname))
2330                         return (zfs_prop_set_userquota(dsname, pair));
2331                 return (-1);
2332         }

2334         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
2335                 nvlist_t *attrs;
2336                 VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
2337                 VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
2338                     &pair) == 0);
2339         }

2341         if (zfs_prop_get_type(prop) == PROP_TYPE_STRING)
2342                 return (-1);

2344         VERIFY(0 == nvpair_value_uint64(pair, &intval));

2346         switch (prop) {
2347         case ZFS_PROP_QUOTA:
2348                 err = dsl_dir_set_quota(dsname, source, intval);
2349                 break;
2350         case ZFS_PROP_REFQUOTA:
2351                 err = dsl_dataset_set_refquota(dsname, source, intval);
2352                 break;
2353         case ZFS_PROP_RESERVATION:
2354                 err = dsl_dir_set_reservation(dsname, source, intval);
2355                 break;
2356         case ZFS_PROP_REFRESERVATION:
2357                 err = dsl_dataset_set_refreservation(dsname, source, intval);
2358                 break;
2359         case ZFS_PROP_VOLSIZE:
2360                 err = zvol_set_volsize(dsname, intval);
2361                 break;
2362         case ZFS_PROP_VERSION:
2363         {
2364                 zfsvfs_t *zfsvfs;

2366                 if ((err = zfsvfs_hold(dsname, FTAG, &zfsvfs, B_TRUE)) != 0)
2367                         break;

2369                 err = zfs_set_version(zfsvfs, intval);
2370                 zfsvfs_rele(zfsvfs, FTAG);

2372                 if (err == 0 && intval >= ZPL_VERSION_USERSPACE) {
2373                         zfs_cmd_t *zc;

2375                         zc = kmem_zalloc(sizeof (zfs_cmd_t), KM_SLEEP);
2376                         (void) strcpy(zc->zc_name, dsname);
2377                         (void) zfs_ioc_userspace_upgrade(zc);
2378                         kmem_free(zc, sizeof (zfs_cmd_t));
2379                 }
2380                 break;
2381         }
```

```
2382         case ZFS_PROP_COMPRESSION:
2383         {
2384                 if (intval == ZIO_COMPRESS_LZ4) {
2385                         zfeature_info_t *feature =
2386                             &spa_feature_table[SPA_FEATURE_LZ4_COMPRESS];
2385                         spa_t *spa;

2387                         if ((err = spa_open(dsname, &spa, FTAG)) != 0)
2388                                 return (err);

2390                         /*
2391                          * Setting the LZ4 compression algorithm activates
2392                          * the feature.
2393                          */
2394                         if (!spa_feature_is_active(spa,
2395                             SPA_FEATURE_LZ4_COMPRESS)) {
2396                         if (!spa_feature_is_active(spa, feature)) {
2396                                 if ((err = zfs_prop_activate_feature(spa,
2397                                     SPA_FEATURE_LZ4_COMPRESS)) != 0) {
2398                                     feature)) != 0) {
2398                                         spa_close(spa, FTAG);
2399                                         return (err);
2400                                 }
2401                         }

2403                         spa_close(spa, FTAG);
2404                 }
2405                 /*
2406                  * We still want the default set action to be performed in the
2407                  * caller, we only performed zfeature settings here.
2408                  */
2409                 err = -1;
2410                 break;
2411         }

2413         default:
2414                 err = -1;
2415         }

2417         return (err);
2418 }
```

*unchanged_portion_omitted_*

```
3583 static int
3584 zfs_check_settable(const char *dsname, nvpair_t *pair, cred_t *cr)
3585 {
3586         const char *propname = nvpair_name(pair);
3587         boolean_t issnap = (strchr(dsname, '@') != NULL);
3588         zfs_prop_t prop = zfs_name_to_prop(propname);
3589         uint64_t intval;
3590         int err;

3592         if (prop == ZPROP_INVAL) {
3593                 if (zfs_prop_user(propname)) {
3594                         if (err = zfs_secpolicy_write_perms(dsname,
3595                             ZFS_DELEG_PERM_USERPROP, cr))
3596                                 return (err);
3597                         return (0);
3598                 }

3600                 if (!issnap && zfs_prop_userquota(propname)) {
3601                         const char *perm = NULL;
3602                         const char *uq_prefix =
3603                             zfs_userquota_prop_prefixes[ZFS_PROP_USERQUOTA];
3604                         const char *gq_prefix =
3605                             zfs_userquota_prop_prefixes[ZFS_PROP_GROUPQUOTA];
```



```
3607                         if (strncmp(propname, uq_prefix,
3608                             strlen(uq_prefix)) == 0) {
3609                                 perm = ZFS_DELEG_PERM_USERQUOTA;
3610                         } else if (strncmp(propname, gq_prefix,
3611                             strlen(gq_prefix)) == 0) {
3612                                 perm = ZFS_DELEG_PERM_GROUPQUOTA;
3613                         } else {
3614                                 /* USERUSED and GROUPUSED are read-only */
3615                                 return (SET_ERROR(EINVAL));
3616                         }

3618                         if (err = zfs_secpolicy_write_perms(dsname, perm, cr))
3619                                 return (err);
3620                         return (0);
3621                 }

3623                 return (SET_ERROR(EINVAL));
3624         }

3626         if (issnap)
3627                 return (SET_ERROR(EINVAL));

3629         if (nvpair_type(pair) == DATA_TYPE_NVLIST) {
3630                 /*
3631                  * dsl_prop_get_all_impl() returns properties in this
3632                  * format.
3633                  */
3634                 nvlist_t *attrs;
3635                 VERIFY(nvpair_value_nvlist(pair, &attrs) == 0);
3636                 VERIFY(nvlist_lookup_nvpair(attrs, ZPROP_VALUE,
3637                     &pair) == 0);
3638         }

3640         /*
3641          * Check that this value is valid for this pool version
3642          */
3643         switch (prop) {
3644         case ZFS_PROP_COMPRESSION:
3645                 /*
3646                  * If the user specified gzip compression, make sure
3647                  * the SPA supports it. We ignore any errors here since
3648                  * we'll catch them later.
3649                  */
3650                 if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3651                     nvpair_value_uint64(pair, &intval) == 0) {
3652                         if (intval >= ZIO_COMPRESS_GZIP_1 &&
3653                             intval <= ZIO_COMPRESS_GZIP_9 &&
3654                             zfs_earlier_version(dsname,
3655                             SPA_VERSION_GZIP_COMPRESSION)) {
3656                                 return (SET_ERROR(ENOTSUP));
3657                         }

3659                         if (intval == ZIO_COMPRESS_ZLE &&
3660                             zfs_earlier_version(dsname,
3661                             SPA_VERSION_ZLE_COMPRESSION))
3662                                 return (SET_ERROR(ENOTSUP));

3664                         if (intval == ZIO_COMPRESS_LZ4) {
3666                                 zfeature_info_t *feature =
3667                                     &spa_feature_table[
3668                                     SPA_FEATURE_LZ4_COMPRESS];
3665                                 spa_t *spa;

3667                                 if ((err = spa_open(dsname, &spa, FTAG)) != 0)
3668                                         return (err);
```

```
3670                                    if (!spa_feature_is_enabled(spa,
3671                                        SPA_FEATURE_LZ4_COMPRESS)) {
3674                                    if (!spa_feature_is_enabled(spa, feature)) {
3672                                            spa_close(spa, FTAG);
3673                                            return (SET_ERROR(ENOTSUP));
3674                                    }
3675                                    spa_close(spa, FTAG);
3676                            }

3678                            /*
3679                             * If this is a bootable dataset then
3680                             * verify that the compression algorithm
3681                             * is supported for booting. We must return
3682                             * something other than ENOTSUP since it
3683                             * implies a downrev pool version.
3684                             */
3685                            if (zfs_is_bootfs(dsname) &&
3686                                !BOOTFS_COMPRESS_VALID(intval)) {
3687                                    return (SET_ERROR(ERANGE));
3688                            }
3689                    }
3690                    break;

3692            case ZFS_PROP_COPIES:
3693                    if (zfs_earlier_version(dsname, SPA_VERSION_DITTO_BLOCKS))
3694                            return (SET_ERROR(ENOTSUP));
3695                    break;

3697            case ZFS_PROP_DEDUP:
3698                    if (zfs_earlier_version(dsname, SPA_VERSION_DEDUP))
3699                            return (SET_ERROR(ENOTSUP));
3700                    break;

3702            case ZFS_PROP_SHARESMB:
3703                    if (zpl_earlier_version(dsname, ZPL_VERSION_FUID))
3704                            return (SET_ERROR(ENOTSUP));
3705                    break;

3707            case ZFS_PROP_ACLINHERIT:
3708                    if (nvpair_type(pair) == DATA_TYPE_UINT64 &&
3709                        nvpair_value_uint64(pair, &intval) == 0) {
3710                            if (intval == ZFS_ACL_PASSTHROUGH_X &&
3711                                zfs_earlier_version(dsname,
3712                                SPA_VERSION_PASSTHROUGH_X))
3713                                    return (SET_ERROR(ENOTSUP));
3714                    }
3715                    break;
3716            }

3718            return (zfs_secpolicy_setprop(dsname, prop, pair, CRED()));
3719    }

3721    /*
3722     * Checks for a race condition to make sure we don't increment a feature flag
3723     * multiple times.
3724     */
3725    static int
3726    zfs_prop_activate_feature_check(void *arg, dmu_tx_t *tx)
3727    {
3728            spa_t *spa = dmu_tx_pool(tx)->dp_spa;
3729            spa_feature_t *featurep = arg;
3732            zfeature_info_t *feature = arg;

3731            if (!spa_feature_is_active(spa, *featurep))
3734            if (!spa_feature_is_active(spa, feature))
```



```
3732                    return (0);
3733            else
3734                    return (SET_ERROR(EBUSY));
3735    }

3737    /*
3738     * The callback invoked on feature activation in the sync task caused by
3739     * zfs_prop_activate_feature.
3740     */
3741    static void
3742    zfs_prop_activate_feature_sync(void *arg, dmu_tx_t *tx)
3743    {
3744            spa_t *spa = dmu_tx_pool(tx)->dp_spa;
3745            spa_feature_t *featurep = arg;
3748            zfeature_info_t *feature = arg;

3747            spa_feature_incr(spa, *featurep, tx);
3750            spa_feature_incr(spa, feature, tx);
3748    }

3750    /*
3751     * Activates a feature on a pool in response to a property setting. This
3752     * creates a new sync task which modifies the pool to reflect the feature
3753     * as being active.
3754     */
3755    static int
3756    zfs_prop_activate_feature(spa_t *spa, spa_feature_t feature)
3759    zfs_prop_activate_feature(spa_t *spa, zfeature_info_t *feature)
3757    {
3758            int err;

3760            /* EBUSY here indicates that the feature is already active */
3761            err = dsl_sync_task(spa_name(spa),
3762                zfs_prop_activate_feature_check, zfs_prop_activate_feature_sync,
3763                &feature, 2);
3766                feature, 2);

3765            if (err != 0 && err != EBUSY)
3766                    return (err);
3767            else
3768                    return (0);
3769    }
_____unchanged_portion_omitted_
```

```
**********************************************************
   52748 Tue Oct  1 14:04:50 2013
new/usr/src/uts/common/fs/zfs/zvol.c
4171 clean up spa_feature_*() interfaces
4172 implement extensible_dataset feature for use by other zpool features
Reviewed by: Max Grossman <max.grossman@delphix.com>
Reviewed by: Christopher Siden <christopher.siden@delphix.com>
Reviewed by: George Wilson <george.wilson@delphix.com>
**********************************************************
______unchanged_portion_omitted_

1831 /*ARGSUSED*/
1832 static int
1833 zfs_mvdev_dump_feature_check(void *arg, dmu_tx_t *tx)
1834 {
1835         spa_t *spa = dmu_tx_pool(tx)->dp_spa;

1837         if (spa_feature_is_active(spa, SPA_FEATURE_MULTI_VDEV_CRASH_DUMP))
1837         if (spa_feature_is_active(spa,
1838             &spa_feature_table[SPA_FEATURE_MULTI_VDEV_CRASH_DUMP]))
1838                 return (1);
1839         return (0);
1840 }

1842 /*ARGSUSED*/
1843 static void
1844 zfs_mvdev_dump_activate_feature_sync(void *arg, dmu_tx_t *tx)
1845 {
1846         spa_t *spa = dmu_tx_pool(tx)->dp_spa;

1848         spa_feature_incr(spa, SPA_FEATURE_MULTI_VDEV_CRASH_DUMP, tx);
1849         spa_feature_incr(spa,
1850             &spa_feature_table[SPA_FEATURE_MULTI_VDEV_CRASH_DUMP], tx);
1849 }

1851 static int
1852 zvol_dump_init(zvol_state_t *zv, boolean_t resize)
1853 {
1854         dmu_tx_t *tx;
1855         int error;
1856         objset_t *os = zv->zv_objset;
1857         spa_t *spa = dmu_objset_spa(os);
1858         vdev_t *vd = spa->spa_root_vdev;
1859         nvlist_t *nv = NULL;
1860         uint64_t version = spa_version(spa);
1861         enum zio_checksum checksum;

1863         ASSERT(MUTEX_HELD(&zfsdev_state_lock));
1864         ASSERT(vd->vdev_ops == &vdev_root_ops);

1866         error = dmu_free_long_range(zv->zv_objset, ZVOL_OBJ, 0,
1867             DMU_OBJECT_END);
1868         /* wait for dmu_free_long_range to actually free the blocks */
1869         txg_wait_synced(dmu_objset_pool(zv->zv_objset), 0);

1871         /*
1872          * If the pool on which the dump device is being initialized has more
1873          * than one child vdev, check that the MULTI_VDEV_CRASH_DUMP feature is
1874          * enabled.  If so, bump that feature's counter to indicate that the
1875          * feature is active. We also check the vdev type to handle the
1876          * following case:
1877          *   # zpool create test raidz disk1 disk2 disk3
1878          *   Now have spa_root_vdev->vdev_children == 1 (the raidz vdev),
1879          *   the raidz vdev itself has 3 children.
1880          */
1881         if (vd->vdev_children > 1 || vd->vdev_ops == &vdev_raidz_ops) {
```



```
1882                 if (!spa_feature_is_enabled(spa,
1883                     SPA_FEATURE_MULTI_VDEV_CRASH_DUMP))
1885                     &spa_feature_table[SPA_FEATURE_MULTI_VDEV_CRASH_DUMP]))
1884                         return (SET_ERROR(ENOTSUP));
1885                 (void) dsl_sync_task(spa_name(spa),
1886                     zfs_mvdev_dump_feature_check,
1887                     zfs_mvdev_dump_activate_feature_sync, NULL, 2);
1888         }

1890         tx = dmu_tx_create(os);
1891         dmu_tx_hold_zap(tx, ZVOL_ZAP_OBJ, TRUE, NULL);
1892         dmu_tx_hold_bonus(tx, ZVOL_OBJ);
1893         error = dmu_tx_assign(tx, TXG_WAIT);
1894         if (error) {
1895                 dmu_tx_abort(tx);
1896                 return (error);
1897         }

1899         /*
1900          * If MULTI_VDEV_CRASH_DUMP is active, use the NOPARITY checksum
1901          * function.  Otherwise, use the old default -- OFF.
1902          */
1903         checksum = spa_feature_is_active(spa,
1904             SPA_FEATURE_MULTI_VDEV_CRASH_DUMP) ? ZIO_CHECKSUM_NOPARITY :
1905             ZIO_CHECKSUM_OFF;
1906             &spa_feature_table[SPA_FEATURE_MULTI_VDEV_CRASH_DUMP]) ?
1907             ZIO_CHECKSUM_NOPARITY : ZIO_CHECKSUM_OFF;

1907         /*
1908          * If we are resizing the dump device then we only need to
1909          * update the refreservation to match the newly updated
1910          * zvolsize. Otherwise, we save off the original state of the
1911          * zvol so that we can restore them if the zvol is ever undumpified.
1912          */
1913         if (resize) {
1914                 error = zap_update(os, ZVOL_ZAP_OBJ,
1915                     zfs_prop_to_name(ZFS_PROP_REFRESERVATION), 8, 1,
1916                     &zv->zv_volsize, tx);
1917         } else {
1918                 uint64_t checksum, compress, refresrv, vbs, dedup;

1920                 error = dsl_prop_get_integer(zv->zv_name,
1921                     zfs_prop_to_name(ZFS_PROP_COMPRESSION), &compress, NULL);
1922                 error = error ? error : dsl_prop_get_integer(zv->zv_name,
1923                     zfs_prop_to_name(ZFS_PROP_CHECKSUM), &checksum, NULL);
1924                 error = error ? error : dsl_prop_get_integer(zv->zv_name,
1925                     zfs_prop_to_name(ZFS_PROP_REFRESERVATION), &refresrv, NULL);
1926                 error = error ? error : dsl_prop_get_integer(zv->zv_name,
1927                     zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE), &vbs, NULL);
1928                 if (version >= SPA_VERSION_DEDUP) {
1929                         error = error ? error :
1930                             dsl_prop_get_integer(zv->zv_name,
1931                             zfs_prop_to_name(ZFS_PROP_DEDUP), &dedup, NULL);
1932                 }

1934                 error = error ? error : zap_update(os, ZVOL_ZAP_OBJ,
1935                     zfs_prop_to_name(ZFS_PROP_COMPRESSION), 8, 1,
1936                     &compress, tx);
1937                 error = error ? error : zap_update(os, ZVOL_ZAP_OBJ,
1938                     zfs_prop_to_name(ZFS_PROP_CHECKSUM), 8, 1, &checksum, tx);
1939                 error = error ? error : zap_update(os, ZVOL_ZAP_OBJ,
1940                     zfs_prop_to_name(ZFS_PROP_REFRESERVATION), 8, 1,
1941                     &refresrv, tx);
1942                 error = error ? error : zap_update(os, ZVOL_ZAP_OBJ,
1943                     zfs_prop_to_name(ZFS_PROP_VOLBLOCKSIZE), 8, 1,
1944                     &vbs, tx);
```

```
1945                 error = error ? error : dmu_object_set_blocksize(
1946                     os, ZVOL_OBJ, SPA_MAXBLOCKSIZE, 0, tx);
1947                 if (version >= SPA_VERSION_DEDUP) {
1948                         error = error ? error : zap_update(os, ZVOL_ZAP_OBJ,
1949                             zfs_prop_to_name(ZFS_PROP_DEDUP), 8, 1,
1950                             &dedup, tx);
1951                 }
1952                 if (error == 0)
1953                         zv->zv_volblocksize = SPA_MAXBLOCKSIZE;
1954         }
1955         dmu_tx_commit(tx);

1957         /*
1958          * We only need update the zvol's property if we are initializing
1959          * the dump area for the first time.
1960          */
1961         if (!resize) {
1962                 VERIFY(nvlist_alloc(&nv, NV_UNIQUE_NAME, KM_SLEEP) == 0);
1963                 VERIFY(nvlist_add_uint64(nv,
1964                     zfs_prop_to_name(ZFS_PROP_REFRESERVATION), 0) == 0);
1965                 VERIFY(nvlist_add_uint64(nv,
1966                     zfs_prop_to_name(ZFS_PROP_COMPRESSION),
1967                     ZIO_COMPRESS_OFF) == 0);
1968                 VERIFY(nvlist_add_uint64(nv,
1969                     zfs_prop_to_name(ZFS_PROP_CHECKSUM),
1970                     checksum) == 0);
1971                 if (version >= SPA_VERSION_DEDUP) {
1972                         VERIFY(nvlist_add_uint64(nv,
1973                             zfs_prop_to_name(ZFS_PROP_DEDUP),
1974                             ZIO_CHECKSUM_OFF) == 0);
1975                 }

1977                 error = zfs_set_prop_nvlist(zv->zv_name, ZPROP_SRC_LOCAL,
1978                     nv, NULL);
1979                 nvlist_free(nv);

1981                 if (error)
1982                         return (error);
1983         }

1985         /* Allocate the space for the dump */
1986         error = zvol_prealloc(zv);
1987         return (error);
1988 }
```

_____***unchanged_portion_omitted_***