```
**********************************************************
  190649 Tue Apr 21 18:19:16 2015
new/usr/src/uts/common/os/zone.c
patch zone
**********************************************************
______unchanged_portion_omitted_

 527 /*
 528  * ZSD routines.
 529  *
 530  * Zone Specific Data (ZSD) is modeled after Thread Specific Data as
 531  * defined by the pthread_key_create() and related interfaces.
 532  *
 533  * Kernel subsystems may register one or more data items and/or
 534  * callbacks to be executed when a zone is created, shutdown, or
 535  * destroyed.
 536  *
 537  * Unlike the thread counterpart, destructor callbacks will be executed
 538  * even if the data pointer is NULL and/or there are no constructor
 539  * callbacks, so it is the responsibility of such callbacks to check for
 540  * NULL data values if necessary.
 541  *
 542  * The locking strategy and overall picture is as follows:
 543  *
 544  * When someone calls zone_key_create(), a template ZSD entry is added to the
 545  * global list "zsd_registered_keys", protected by zsd_key_lock.  While
 546  * holding that lock all the existing zones are marked as
 547  * ZSD_CREATE_NEEDED and a copy of the ZSD entry added to the per-zone
 548  * zone_zsd list (protected by zone_lock). The global list is updated first
 549  * (under zone_key_lock) to make sure that newly created zones use the
 550  * most recent list of keys. Then under zonehash_lock we walk the zones
 551  * and mark them.  Similar locking is used in zone_key_delete().
 552  *
 553  * The actual create, shutdown, and destroy callbacks are done without
 554  * holding any lock. And zsd_flags are used to ensure that the operations
 555  * completed so that when zone_key_create (and zone_create) is done, as well as
 556  * zone_key_delete (and zone_destroy) is done, all the necessary callbacks
 557  * are completed.
 558  *
 559  * When new zones are created constructor callbacks for all registered ZSD
 560  * entries will be called. That also uses the above two phases of marking
 561  * what needs to be done, and then running the callbacks without holding
 562  * any locks.
 563  *
 564  * The framework does not provide any locking around zone_getspecific() and
 565  * zone_setspecific() apart from that needed for internal consistency, so
 566  * callers interested in atomic "test-and-set" semantics will need to provide
 567  * their own locking.
 568  */

 570 /*
 571  * Helper function to find the zsd_entry associated with the key in the
 572  * given list.
 573  */
 574 static struct zsd_entry *
 575 zsd_find(list_t *l, zone_key_t key)
 576 {
 577         struct zsd_entry *zsd;

 579         list_for_each(l, zsd) {
 579         for (zsd = list_head(l); zsd != NULL; zsd = list_next(l, zsd)) {
 580                 if (zsd->zsd_key == key) {
 581                         return (zsd);
 582                 }
 583         }
 584         return (NULL);
```



```
 585 }

 587 /*
 588  * Helper function to find the zsd_entry associated with the key in the
 589  * given list. Move it to the front of the list.
 590  */
 591 static struct zsd_entry *
 592 zsd_find_mru(list_t *l, zone_key_t key)
 593 {
 594         struct zsd_entry *zsd;

 596         list_for_each(l, zsd) {
 596         for (zsd = list_head(l); zsd != NULL; zsd = list_next(l, zsd)) {
 597                 if (zsd->zsd_key == key) {
 598                         /*
 599                          * Move to head of list to keep list in MRU order.
 600                          */
 601                         if (zsd != list_head(l)) {
 602                                 list_remove(l, zsd);
 603                                 list_insert_head(l, zsd);
 604                         }
 605                         return (zsd);
 606                 }
 607         }
 608         return (NULL);
 609 }

 611 void
 612 zone_key_create(zone_key_t *keyp, void *(*create)(zoneid_t),
 613     void (*shutdown)(zoneid_t, void *), void (*destroy)(zoneid_t, void *))
 614 {
 615         struct zsd_entry *zsdp;
 616         struct zsd_entry *t;
 617         struct zone *zone;
 618         zone_key_t  key;

 620         zsdp = kmem_zalloc(sizeof (*zsdp), KM_SLEEP);
 621         zsdp->zsd_data = NULL;
 622         zsdp->zsd_create = create;
 623         zsdp->zsd_shutdown = shutdown;
 624         zsdp->zsd_destroy = destroy;

 626         /*
 627          * Insert in global list of callbacks. Makes future zone creations
 628          * see it.
 629          */
 630         mutex_enter(&zsd_key_lock);
 631         key = zsdp->zsd_key = ++zsd_keyval;
 632         ASSERT(zsd_keyval != 0);
 633         list_insert_tail(&zsd_registered_keys, zsdp);
 634         mutex_exit(&zsd_key_lock);

 636         /*
 637          * Insert for all existing zones and mark them as needing
 638          * a create callback.
 639          */
 640         mutex_enter(&zonehash_lock);    /* stop the world */
 641         list_for_each(&zone_active, zone) {
 641         for (zone = list_head(&zone_active); zone != NULL;
 642             zone = list_next(&zone_active, zone)) {
 642                 zone_status_t status;

 644                 mutex_enter(&zone->zone_lock);

 646                 /* Skip zones that are on the way down or not yet up */
 647                 status = zone_status_get(zone);
```

```
                if (status >= ZONE_IS_DOWN ||
                    status == ZONE_IS_UNINITIALIZED) {
                        mutex_exit(&zone->zone_lock);
                        continue;
                }

                t = zsd_find_mru(&zone->zone_zsd, key);
                if (t != NULL) {
                        /*
                         * A zsd_configure already inserted it after
                         * we dropped zsd_key_lock above.
                         */
                        mutex_exit(&zone->zone_lock);
                        continue;
                }
                t = kmem_zalloc(sizeof (*t), KM_SLEEP);
                t->zsd_key = key;
                t->zsd_create = create;
                t->zsd_shutdown = shutdown;
                t->zsd_destroy = destroy;
                if (create != NULL) {
                        t->zsd_flags = ZSD_CREATE_NEEDED;
                        DTRACE_PROBE2(zsd__create__needed,
                            zone_t *, zone, zone_key_t, key);
                }
                list_insert_tail(&zone->zone_zsd, t);
                mutex_exit(&zone->zone_lock);
        }
        mutex_exit(&zonehash_lock);

        if (create != NULL) {
                /* Now call the create callback for this key */
                zsd_apply_all_zones(zsd_apply_create, key);
        }
        /*
         * It is safe for consumers to use the key now, make it
         * globally visible. Specifically zone_getspecific() will
         * always successfully return the zone specific data associated
         * with the key.
         */
        *keyp = key;

}

/*
 * Function called when a module is being unloaded, or otherwise wishes
 * to unregister its ZSD key and callbacks.
 *
 * Remove from the global list and determine the functions that need to
 * be called under a global lock. Then call the functions without
 * holding any locks. Finally free up the zone_zsd entries. (The apply
 * functions need to access the zone_zsd entries to find zsd_data etc.)
 */
int
zone_key_delete(zone_key_t key)
{
        struct zsd_entry *zsdp = NULL;
        zone_t *zone;

        mutex_enter(&zsd_key_lock);
        zsdp = zsd_find_mru(&zsd_registered_keys, key);
        if (zsdp == NULL) {
                mutex_exit(&zsd_key_lock);
                return (-1);
        }
        list_remove(&zsd_registered_keys, zsdp);
```



```
        mutex_exit(&zsd_key_lock);

        mutex_enter(&zonehash_lock);
        list_for_each(&zone_active, zone) {
        for (zone = list_head(&zone_active); zone != NULL;
            zone = list_next(&zone_active, zone)) {
                struct zsd_entry *del;

                mutex_enter(&zone->zone_lock);
                del = zsd_find_mru(&zone->zone_zsd, key);
                if (del == NULL) {
                        /*
                         * Somebody else got here first e.g the zone going
                         * away.
                         */
                        mutex_exit(&zone->zone_lock);
                        continue;
                }
                ASSERT(del->zsd_shutdown == zsdp->zsd_shutdown);
                ASSERT(del->zsd_destroy == zsdp->zsd_destroy);
                if (del->zsd_shutdown != NULL &&
                    (del->zsd_flags & ZSD_SHUTDOWN_ALL) == 0) {
                        del->zsd_flags |= ZSD_SHUTDOWN_NEEDED;
                        DTRACE_PROBE2(zsd__shutdown__needed,
                            zone_t *, zone, zone_key_t, key);
                }
                if (del->zsd_destroy != NULL &&
                    (del->zsd_flags & ZSD_DESTROY_ALL) == 0) {
                        del->zsd_flags |= ZSD_DESTROY_NEEDED;
                        DTRACE_PROBE2(zsd__destroy__needed,
                            zone_t *, zone, zone_key_t, key);
                }
                mutex_exit(&zone->zone_lock);
        }
        mutex_exit(&zonehash_lock);
        kmem_free(zsdp, sizeof (*zsdp));

        /* Now call the shutdown and destroy callback for this key */
        zsd_apply_all_zones(zsd_apply_shutdown, key);
        zsd_apply_all_zones(zsd_apply_destroy, key);

        /* Now we can free up the zsdp structures in each zone */
        mutex_enter(&zonehash_lock);
        list_for_each(&zone_active, zone) {
        for (zone = list_head(&zone_active); zone != NULL;
            zone = list_next(&zone_active, zone)) {
                struct zsd_entry *del;

                mutex_enter(&zone->zone_lock);
                del = zsd_find(&zone->zone_zsd, key);
                if (del != NULL) {
                        list_remove(&zone->zone_zsd, del);
                        ASSERT(!(del->zsd_flags & ZSD_ALL_INPROGRESS));
                        kmem_free(del, sizeof (*del));
                }
                mutex_exit(&zone->zone_lock);
        }
        mutex_exit(&zonehash_lock);

        return (0);
}
_____unchanged_portion_omitted_

/*
 * Function used to initialize a zone's list of ZSD callbacks and data
 * when the zone is being created.  The callbacks are initialized from
```

```
 818  * the template list (zsd_registered_keys). The constructor callback is
 819  * executed later (once the zone exists and with locks dropped).
 820  */
 821 static void
 822 zone_zsd_configure(zone_t *zone)
 823 {
 824         struct zsd_entry *zsdp;
 825         struct zsd_entry *t;

 827         ASSERT(MUTEX_HELD(&zonehash_lock));
 828         ASSERT(list_head(&zone->zone_zsd) == NULL);
 829         mutex_enter(&zone->zone_lock);
 830         mutex_enter(&zsd_key_lock);
 831         list_for_each(&zsd_registered_keys, zsdp) {
 834         for (zsdp = list_head(&zsd_registered_keys); zsdp != NULL;
 835             zsdp = list_next(&zsd_registered_keys, zsdp)) {
 832                 /*
 833                  * Since this zone is ZONE_IS_UNCONFIGURED, zone_key_create
 834                  * should not have added anything to it.
 835                  */
 836                 ASSERT(zsd_find(&zone->zone_zsd, zsdp->zsd_key) == NULL);

 838                 t = kmem_zalloc(sizeof (*t), KM_SLEEP);
 839                 t->zsd_key = zsdp->zsd_key;
 840                 t->zsd_create = zsdp->zsd_create;
 841                 t->zsd_shutdown = zsdp->zsd_shutdown;
 842                 t->zsd_destroy = zsdp->zsd_destroy;
 843                 if (zsdp->zsd_create != NULL) {
 844                         t->zsd_flags = ZSD_CREATE_NEEDED;
 845                         DTRACE_PROBE2(zsd__create__needed,
 846                             zone_t *, zone, zone_key_t, zsdp->zsd_key);
 847                 }
 848                 list_insert_tail(&zone->zone_zsd, t);
 849         }
 850         mutex_exit(&zsd_key_lock);
 851         mutex_exit(&zone->zone_lock);
 852 }

 854 enum zsd_callback_type { ZSD_CREATE, ZSD_SHUTDOWN, ZSD_DESTROY };

 856 /*
 857  * Helper function to execute shutdown or destructor callbacks.
 858  */
 859 static void
 860 zone_zsd_callbacks(zone_t *zone, enum zsd_callback_type ct)
 861 {
 862         struct zsd_entry *t;

 864         ASSERT(ct == ZSD_SHUTDOWN || ct == ZSD_DESTROY);
 865         ASSERT(ct != ZSD_SHUTDOWN || zone_status_get(zone) >= ZONE_IS_EMPTY);
 866         ASSERT(ct != ZSD_DESTROY || zone_status_get(zone) >= ZONE_IS_DOWN);

 868         /*
 869          * Run the callback solely based on what is registered for the zone
 870          * in zone_zsd. The global list can change independently of this
 871          * as keys are registered and unregistered and we don't register new
 872          * callbacks for a zone that is in the process of going away.
 873          */
 874         mutex_enter(&zone->zone_lock);
 875         list_for_each(&zone->zone_zsd, t) {
 879         for (t = list_head(&zone->zone_zsd); t != NULL;
 880             t = list_next(&zone->zone_zsd, t)) {
 876                 zone_key_t key = t->zsd_key;

 878                 /* Skip if no callbacks registered */
```



```
 880                 if (ct == ZSD_SHUTDOWN) {
 881                         if (t->zsd_shutdown != NULL &&
 882                             (t->zsd_flags & ZSD_SHUTDOWN_ALL) == 0) {
 883                                 t->zsd_flags |= ZSD_SHUTDOWN_NEEDED;
 884                                 DTRACE_PROBE2(zsd__shutdown__needed,
 885                                     zone_t *, zone, zone_key_t, key);
 886                         }
 887                 } else {
 888                         if (t->zsd_destroy != NULL &&
 889                             (t->zsd_flags & ZSD_DESTROY_ALL) == 0) {
 890                                 t->zsd_flags |= ZSD_DESTROY_NEEDED;
 891                                 DTRACE_PROBE2(zsd__destroy__needed,
 892                                     zone_t *, zone, zone_key_t, key);
 893                         }
 894                 }
 895         }
 896         mutex_exit(&zone->zone_lock);

 898         /* Now call the shutdown and destroy callback for this key */
 899         zsd_apply_all_keys(zsd_apply_shutdown, zone);
 900         zsd_apply_all_keys(zsd_apply_destroy, zone);

 902 }

 904 /*
 905  * Called when the zone is going away; free ZSD-related memory, and
 906  * destroy the zone_zsd list.
 907  */
 908 static void
 909 zone_free_zsd(zone_t *zone)
 910 {
 911         struct zsd_entry *t, *next;

 913         /*
 914          * Free all the zsd_entry's we had on this zone.
 915          */
 916         mutex_enter(&zone->zone_lock);
 917         list_for_each_safe(&zone->zone_zsd, t, next) {
 922         for (t = list_head(&zone->zone_zsd); t != NULL; t = next) {
 923                 next = list_next(&zone->zone_zsd, t);
 918                 list_remove(&zone->zone_zsd, t);
 919                 ASSERT(!(t->zsd_flags & ZSD_ALL_INPROGRESS));
 920                 kmem_free(t, sizeof (*t));
 921         }
 922         list_destroy(&zone->zone_zsd);
 923         mutex_exit(&zone->zone_lock);

 925 }
_____unchanged_portion_omitted_

1290 /*
1291  * Frees memory associated with the zone dataset list.
1292  */
1293 static void
1294 zone_free_datasets(zone_t *zone)
1295 {
1296         zone_dataset_t *t, *next;

1298         list_for_each_safe(&zone->zone_datasets, t, next) {
1304         for (t = list_head(&zone->zone_datasets); t != NULL; t = next) {
1305                 next = list_next(&zone->zone_datasets, t);
1299                 list_remove(&zone->zone_datasets, t);
1300                 kmem_free(t->zd_dataset, strlen(t->zd_dataset) + 1);
1301                 kmem_free(t, sizeof (*t));
1302         }
1303         list_destroy(&zone->zone_datasets);
```

```
1304 }
_____unchanged_portion_omitted_

3007 /*
3008  * Similar to zone_find_by_id(), using the path as a key.  For instance,
3009  * if there is a zone "foo" rooted at /foo/root, and the path argument
3010  * is "/foo/root/proc", it will return the held zone_t corresponding to
3011  * zone "foo".
3012  *
3013  * zone_find_by_path() always returns a non-NULL value, since at the
3014  * very least every path will be contained in the global zone.
3015  *
3016  * As with the other zone_find_by_*() functions, the caller is
3017  * responsible for zone_rele()ing the return value of this function.
3018  */
3019 zone_t *
3020 zone_find_by_path(const char *path)
3021 {
3022         zone_t *zone;
3023         zone_t *zret = NULL;
3024         zone_status_t status;

3026         if (path == NULL) {
3027                 /*
3028                  * Call from rootconf().
3029                  */
3030                 zone_hold(global_zone);
3031                 return (global_zone);
3032         }
3033         ASSERT(*path == '/');
3034         mutex_enter(&zonehash_lock);
3035         list_for_each(&zone_active, zone) {
3042         for (zone = list_head(&zone_active); zone != NULL;
3043             zone = list_next(&zone_active, zone)) {
3036                 if (ZONE_PATH_VISIBLE(path, zone))
3037                         zret = zone;
3038         }
3039         ASSERT(zret != NULL);
3040         status = zone_status_get(zret);
3041         if (status < ZONE_IS_READY || status > ZONE_IS_DOWN) {
3042                 /*
3043                  * Zone practically doesn't exist.
3044                  */
3045                 zret = global_zone;
3046         }
3047         zone_hold(zret);
3048         mutex_exit(&zonehash_lock);
3049         return (zret);
3050 }
_____unchanged_portion_omitted_

3237 /*
3238  * Walk the list of active zones and issue the provided callback for
3239  * each of them.
3240  *
3241  * Caller must not be holding any locks that may be acquired under
3242  * zonehash_lock.  See comment at the beginning of the file for a list of
3243  * common locks and their interactions with zones.
3244  */
3245 int
3246 zone_walk(int (*cb)(zone_t *, void *), void *data)
3247 {
3248         zone_t *zone;
3249         int ret = 0;
3250         zone_status_t status;
```



```
3252         mutex_enter(&zonehash_lock);
3253         list_for_each(&zone_active, zone) {
3261         for (zone = list_head(&zone_active); zone != NULL;
3262             zone = list_next(&zone_active, zone)) {
3254                 /*
3255                  * Skip zones that shouldn't be externally visible.
3256                  */
3257                 status = zone_status_get(zone);
3258                 if (status < ZONE_IS_READY || status > ZONE_IS_DOWN)
3259                         continue;
3260                 /*
3261                  * Bail immediately if any callback invocation returns a
3262                  * non-zero value.
3263                  */
3264                 ret = (*cb)(zone, data);
3265                 if (ret != 0)
3266                         break;
3267         }
3268         mutex_exit(&zonehash_lock);
3269         return (ret);
3270 }
_____unchanged_portion_omitted_

4024 /*
4025  * Helper function to make sure that a zone created on 'rootpath'
4026  * wouldn't end up containing other zones' rootpaths.
4027  */
4028 static boolean_t
4029 zone_is_nested(const char *rootpath)
4030 {
4031         zone_t *zone;
4032         size_t rootpathlen = strlen(rootpath);
4033         size_t len;

4035         ASSERT(MUTEX_HELD(&zonehash_lock));

4037         /*
4038          * zone_set_root() appended '/' and '\0' at the end of rootpath
4039          */
4040         if ((rootpathlen <= 3) && (rootpath[0] == '/') &&
4041             (rootpath[1] == '/') && (rootpath[2] == '\0'))
4042                 return (B_TRUE);

4044         list_for_each(&zone_active, zone) {
4053         for (zone = list_head(&zone_active); zone != NULL;
4054             zone = list_next(&zone_active, zone)) {
4045                 if (zone == global_zone)
4046                         continue;
4047                 len = strlen(zone->zone_rootpath);
4048                 if (strncmp(rootpath, zone->zone_rootpath,
4049                     MIN(rootpathlen, len)) == 0)
4050                         return (B_TRUE);
4051         }
4052         return (B_FALSE);
4053 }
_____unchanged_portion_omitted_

5624 /*
5625  * Systemcall entry point for zone_enter().
5626  *
5627  * The current process is injected into said zone.  In the process
5628  * it will change its project membership, privileges, rootdir/cwd,
5629  * zone-wide rctls, and pool association to match those of the zone.
5630  *
5631  * The first zone_enter() called while the zone is in the ZONE_IS_READY
5632  * state will transition it to ZONE_IS_RUNNING.  Processes may only
```

```
5633  * enter a zone that is "ready" or "running".
5634  */
5635 static int
5636 zone_enter(zoneid_t zoneid)
5637 {
5638         zone_t *zone;
5639         vnode_t *vp;
5640         proc_t *pp = curproc;
5641         contract_t *ct;
5642         cont_process_t *ctp;
5643         task_t *tk, *oldtk;
5644         kproject_t *zone_proj0;
5645         cred_t *cr, *newcr;
5646         pool_t *oldpool, *newpool;
5647         sess_t *sp;
5648         uid_t uid;
5649         zone_status_t status;
5650         int err = 0;
5651         rctl_entity_p_t e;
5652         size_t swap;
5653         kthread_id_t t;

5655         if (secpolicy_zone_config(CRED()) != 0)
5656                 return (set_errno(EPERM));
5657         if (zoneid < MIN_USERZONEID || zoneid > MAX_ZONEID)
5658                 return (set_errno(EINVAL));

5660         /*
5661          * Stop all lwps so we don't need to hold a lock to look at
5662          * curproc->p_zone.  This needs to happen before we grab any
5663          * locks to avoid deadlock (another lwp in the process could
5664          * be waiting for the held lock).
5665          */
5666         if (curthread != pp->p_agenttp && !holdlwps(SHOLDFORK))
5667                 return (set_errno(EINTR));

5669         /*
5670          * Make sure we're not changing zones with files open or mapped in
5671          * to our address space which shouldn't be changing zones.
5672          */
5673         if (!files_can_change_zones()) {
5674                 err = EBADF;
5675                 goto out;
5676         }
5677         if (!as_can_change_zones()) {
5678                 err = EFAULT;
5679                 goto out;
5680         }

5682         mutex_enter(&zonehash_lock);
5683         if (pp->p_zone != global_zone) {
5684                 mutex_exit(&zonehash_lock);
5685                 err = EINVAL;
5686                 goto out;
5687         }

5689         zone = zone_find_all_by_id(zoneid);
5690         if (zone == NULL) {
5691                 mutex_exit(&zonehash_lock);
5692                 err = EINVAL;
5693                 goto out;
5694         }

5696         /*
5697          * To prevent processes in a zone from holding contracts on
5698          * extrazonal resources, and to avoid process contract
```



```
5699          * memberships which span zones, contract holders and processes
5700          * which aren't the sole members of their encapsulating process
5701          * contracts are not allowed to zone_enter.
5702          */
5703         ctp = pp->p_ct_process;
5704         ct = &ctp->conp_contract;
5705         mutex_enter(&ct->ct_lock);
5706         mutex_enter(&pp->p_lock);
5707         if ((avl_numnodes(&pp->p_ct_held) != 0) || (ctp->conp_nmembers != 1)) {
5708                 mutex_exit(&pp->p_lock);
5709                 mutex_exit(&ct->ct_lock);
5710                 mutex_exit(&zonehash_lock);
5711                 err = EINVAL;
5712                 goto out;
5713         }

5715         /*
5716          * Moreover, we don't allow processes whose encapsulating
5717          * process contracts have inherited extrazonal contracts.
5718          * While it would be easier to eliminate all process contracts
5719          * with inherited contracts, we need to be able to give a
5720          * restarted init (or other zone-penetrating process) its
5721          * predecessor's contracts.
5722          */
5723         if (ctp->conp_ninherited != 0) {
5724                 contract_t *next;
5725                 list_for_each(&ctp->conp_inherited, next) {
5735                 for (next = list_head(&ctp->conp_inherited); next;
5736                     next = list_next(&ctp->conp_inherited, next)) {
5726                         if (contract_getzuniqid(next) != zone->zone_uniqid) {
5727                                 mutex_exit(&pp->p_lock);
5728                                 mutex_exit(&ct->ct_lock);
5729                                 mutex_exit(&zonehash_lock);
5730                                 err = EINVAL;
5731                                 goto out;
5732                         }
5733                 }
5734         }

5736         mutex_exit(&pp->p_lock);
5737         mutex_exit(&ct->ct_lock);

5739         status = zone_status_get(zone);
5740         if (status < ZONE_IS_READY || status >= ZONE_IS_SHUTTING_DOWN) {
5741                 /*
5742                  * Can't join
5743                  */
5744                 mutex_exit(&zonehash_lock);
5745                 err = EINVAL;
5746                 goto out;
5747         }

5749         /*
5750          * Make sure new priv set is within the permitted set for caller
5751          */
5752         if (!priv_issubset(zone->zone_privset, &CR_OPPRIV(CRED()))) {
5753                 mutex_exit(&zonehash_lock);
5754                 err = EPERM;
5755                 goto out;
5756         }
5757         /*
5758          * We want to momentarily drop zonehash_lock while we optimistically
5759          * bind curproc to the pool it should be running in.  This is safe
5760          * since the zone can't disappear (we have a hold on it).
5761          */
5762         zone_hold(zone);
```

```
5763         mutex_exit(&zonehash_lock);

5765         /*
5766          * Grab pool_lock to keep the pools configuration from changing
5767          * and to stop ourselves from getting rebound to another pool
5768          * until we join the zone.
5769          */
5770         if (pool_lock_intr() != 0) {
5771                 zone_rele(zone);
5772                 err = EINTR;
5773                 goto out;
5774         }
5775         ASSERT(secpolicy_pool(CRED()) == 0);
5776         /*
5777          * Bind ourselves to the pool currently associated with the zone.
5778          */
5779         oldpool = curproc->p_pool;
5780         newpool = zone_pool_get(zone);
5781         if (pool_state == POOL_ENABLED && newpool != oldpool &&
5782             (err = pool_do_bind(newpool, P_PID, P_MYID,
5783             POOL_BIND_ALL)) != 0) {
5784                 pool_unlock();
5785                 zone_rele(zone);
5786                 goto out;
5787         }

5789         /*
5790          * Grab cpu_lock now; we'll need it later when we call
5791          * task_join().
5792          */
5793         mutex_enter(&cpu_lock);
5794         mutex_enter(&zonehash_lock);
5795         /*
5796          * Make sure the zone hasn't moved on since we dropped zonehash_lock.
5797          */
5798         if (zone_status_get(zone) >= ZONE_IS_SHUTTING_DOWN) {
5799                 /*
5800                  * Can't join anymore.
5801                  */
5802                 mutex_exit(&zonehash_lock);
5803                 mutex_exit(&cpu_lock);
5804                 if (pool_state == POOL_ENABLED &&
5805                     newpool != oldpool)
5806                         (void) pool_do_bind(oldpool, P_PID, P_MYID,
5807                             POOL_BIND_ALL);
5808                 pool_unlock();
5809                 zone_rele(zone);
5810                 err = EINVAL;
5811                 goto out;
5812         }

5814         /*
5815          * a_lock must be held while transfering locked memory and swap
5816          * reservation from the global zone to the non global zone because
5817          * asynchronous faults on the processes' address space can lock
5818          * memory and reserve swap via MCL_FUTURE and MAP_NORESERVE
5819          * segments respectively.
5820          */
5821         AS_LOCK_ENTER(pp->as, &pp->p_as->a_lock, RW_WRITER);
5822         swap = as_swresv();
5823         mutex_enter(&pp->p_lock);
5824         zone_proj0 = zone->zone_zsched->p_task->tk_proj;
5825         /* verify that we do not exceed and task or lwp limits */
5826         mutex_enter(&zone->zone_nlwps_lock);
5827         /* add new lwps to zone and zone's proj0 */
5828         zone_proj0->kpj_nlwps += pp->p_lwpcnt;
```



```
5829         zone->zone_nlwps += pp->p_lwpcnt;
5830         /* add 1 task to zone's proj0 */
5831         zone_proj0->kpj_ntasks += 1;

5833         zone_proj0->kpj_nprocs++;
5834         zone->zone_nprocs++;
5835         mutex_exit(&zone->zone_nlwps_lock);

5837         mutex_enter(&zone->zone_mem_lock);
5838         zone->zone_locked_mem += pp->p_locked_mem;
5839         zone_proj0->kpj_data.kpd_locked_mem += pp->p_locked_mem;
5840         zone->zone_max_swap += swap;
5841         mutex_exit(&zone->zone_mem_lock);

5843         mutex_enter(&(zone_proj0->kpj_data.kpd_crypto_lock));
5844         zone_proj0->kpj_data.kpd_crypto_mem += pp->p_crypto_mem;
5845         mutex_exit(&(zone_proj0->kpj_data.kpd_crypto_lock));

5847         /* remove lwps and process from proc's old zone and old project */
5848         mutex_enter(&pp->p_zone->zone_nlwps_lock);
5849         pp->p_zone->zone_nlwps -= pp->p_lwpcnt;
5850         pp->p_task->tk_proj->kpj_nlwps -= pp->p_lwpcnt;
5851         pp->p_task->tk_proj->kpj_nprocs--;
5852         pp->p_zone->zone_nprocs--;
5853         mutex_exit(&pp->p_zone->zone_nlwps_lock);

5855         mutex_enter(&pp->p_zone->zone_mem_lock);
5856         pp->p_zone->zone_locked_mem -= pp->p_locked_mem;
5857         pp->p_task->tk_proj->kpj_data.kpd_locked_mem -= pp->p_locked_mem;
5858         pp->p_zone->zone_max_swap -= swap;
5859         mutex_exit(&pp->p_zone->zone_mem_lock);

5861         mutex_enter(&(pp->p_task->tk_proj->kpj_data.kpd_crypto_lock));
5862         pp->p_task->tk_proj->kpj_data.kpd_crypto_mem -= pp->p_crypto_mem;
5863         mutex_exit(&(pp->p_task->tk_proj->kpj_data.kpd_crypto_lock));

5865         pp->p_flag |= SZONETOP;
5866         pp->p_zone = zone;
5867         mutex_exit(&pp->p_lock);
5868         AS_LOCK_EXIT(pp->p_as, &pp->p_as->a_lock);

5870         /*
5871          * Joining the zone cannot fail from now on.
5872          *
5873          * This means that a lot of the following code can be commonized and
5874          * shared with zsched().
5875          */

5877         /*
5878          * If the process contract fmri was inherited, we need to
5879          * flag this so that any contract status will not leak
5880          * extra zone information, svc_fmri in this case
5881          */
5882         if (ctp->conp_svc_ctid != ct->ct_id) {
5883                 mutex_enter(&ct->ct_lock);
5884                 ctp->conp_svc_zone_enter = ct->ct_id;
5885                 mutex_exit(&ct->ct_lock);
5886         }

5888         /*
5889          * Reset the encapsulating process contract's zone.
5890          */
5891         ASSERT(ct->ct_mzuniqid == GLOBAL_ZONEUNIQID);
5892         contract_setzuniqid(ct, zone->zone_uniqid);

5894         /*
```

```
5895          * Create a new task and associate the process with the project keyed
5896          * by (projid,zoneid).
5897          *
5898          * We might as well be in project 0; the global zone's projid doesn't
5899          * make much sense in a zone anyhow.
5900          *
5901          * This also increments zone_ntasks, and returns with p_lock held.
5902          */
5903         tk = task_create(0, zone);
5904         oldtk = task_join(tk, 0);
5905         mutex_exit(&cpu_lock);

5907         /*
5908          * call RCTLOP_SET functions on this proc
5909          */
5910         e.rcep_p.zone = zone;
5911         e.rcep_t = RCENTITY_ZONE;
5912         (void) rctl_set_dup(NULL, NULL, pp, &e, zone->zone_rctls, NULL,
5913             RCD_CALLBACK);
5914         mutex_exit(&pp->p_lock);

5916         /*
5917          * We don't need to hold any of zsched's locks here; not only do we know
5918          * the process and zone aren't going away, we know its session isn't
5919          * changing either.
5920          *
5921          * By joining zsched's session here, we mimic the behavior in the
5922          * global zone of init's sid being the pid of sched.  We extend this
5923          * to all zlogin-like zone_enter()'ing processes as well.
5924          */
5925         mutex_enter(&pidlock);
5926         sp = zone->zone_zsched->p_sessp;
5927         sess_hold(zone->zone_zsched);
5928         mutex_enter(&pp->p_lock);
5929         pgexit(pp);
5930         sess_rele(pp->p_sessp, B_TRUE);
5931         pp->p_sessp = sp;
5932         pgjoin(pp, zone->zone_zsched->p_pidp);

5934         /*
5935          * If any threads are scheduled to be placed on zone wait queue they
5936          * should abandon the idea since the wait queue is changing.
5937          * We need to be holding pidlock & p_lock to do this.
5938          */
5939         if ((t = pp->p_tlist) != NULL) {
5940                 do {
5941                         thread_lock(t);
5942                         /*
5943                          * Kick this thread so that he doesn't sit
5944                          * on a wrong wait queue.
5945                          */
5946                         if (ISWAITING(t))
5947                                 setrun_locked(t);

5949                         if (t->t_schedflag & TS_ANYWAITQ)
5950                                 t->t_schedflag &= ~ TS_ANYWAITQ;

5952                         thread_unlock(t);
5953                 } while ((t = t->t_forw) != pp->p_tlist);
5954         }

5956         /*
5957          * If there is a default scheduling class for the zone and it is not
5958          * the class we are currently in, change all of the threads in the
5959          * process to the new class.  We need to be holding pidlock & p_lock
5960          * when we call parmsset so this is a good place to do it.
```



```
5961          */
5962         if (zone->zone_defaultcid > 0 &&
5963             zone->zone_defaultcid != curthread->t_cid) {
5964                 pcparms_t pcparms;

5966                 pcparms.pc_cid = zone->zone_defaultcid;
5967                 pcparms.pc_clparms[0] = 0;

5969                 /*
5970                  * If setting the class fails, we still want to enter the zone.
5971                  */
5972                 if ((t = pp->p_tlist) != NULL) {
5973                         do {
5974                                 (void) parmsset(&pcparms, t);
5975                         } while ((t = t->t_forw) != pp->p_tlist);
5976                 }
5977         }

5979         mutex_exit(&pp->p_lock);
5980         mutex_exit(&pidlock);

5982         mutex_exit(&zonehash_lock);
5983         /*
5984          * We're firmly in the zone; let pools progress.
5985          */
5986         pool_unlock();
5987         task_rele(oldtk);
5988         /*
5989          * We don't need to retain a hold on the zone since we already
5990          * incremented zone_ntasks, so the zone isn't going anywhere.
5991          */
5992         zone_rele(zone);

5994         /*
5995          * Chroot
5996          */
5997         vp = zone->zone_rootvp;
5998         zone_chdir(vp, &PTOU(pp)->u_cdir, pp);
5999         zone_chdir(vp, &PTOU(pp)->u_rdir, pp);

6001         /*
6002          * Change process credentials
6003          */
6004         newcr = cralloc();
6005         mutex_enter(&pp->p_crlock);
6006         cr = pp->p_cred;
6007         crcopy_to(cr, newcr);
6008         crsetzone(newcr, zone);
6009         pp->p_cred = newcr;

6011         /*
6012          * Restrict all process privilege sets to zone limit
6013          */
6014         priv_intersect(zone->zone_privset, &CR_PPRIV(newcr));
6015         priv_intersect(zone->zone_privset, &CR_EPRIV(newcr));
6016         priv_intersect(zone->zone_privset, &CR_IPRIV(newcr));
6017         priv_intersect(zone->zone_privset, &CR_LPRIV(newcr));
6018         mutex_exit(&pp->p_crlock);
6019         crset(pp, newcr);

6021         /*
6022          * Adjust upcount to reflect zone entry.
6023          */
6024         uid = crgetruid(newcr);
6025         mutex_enter(&pidlock);
6026         upcount_dec(uid, GLOBAL_ZONEID);
```

```
6027         upcount_inc(uid, zoneid);
6028         mutex_exit(&pidlock);
6029
6030         /*
6031          * Set up core file path and content.
6032          */
6033         set_core_defaults();
6034
6035 out:
6036         /*
6037          * Let the other lwps continue.
6038          */
6039         mutex_enter(&pp->p_lock);
6040         if (curthread != pp->p_agenttp)
6041                 continuelwps(pp);
6042         mutex_exit(&pp->p_lock);
6043
6044         return (err != 0 ? set_errno(err) : 0);
6045 }
6046
6047 /*
6048  * Systemcall entry point for zone_list(2).
6049  *
6050  * Processes running in a (non-global) zone only see themselves.
6051  * On labeled systems, they see all zones whose label they dominate.
6052  */
6053 static int
6054 zone_list(zoneid_t *zoneidlist, uint_t *numzones)
6055 {
6056         zoneid_t *zoneids;
6057         zone_t *zone, *myzone;
6058         uint_t user_nzones, real_nzones;
6059         uint_t domi_nzones;
6060         int error;
6061
6062         if (copyin(numzones, &user_nzones, sizeof (uint_t)) != 0)
6063                 return (set_errno(EFAULT));
6064
6065         myzone = curproc->p_zone;
6066         if (myzone != global_zone) {
6067                 bslabel_t *mybslab;
6068
6069                 if (!is_system_labeled()) {
6070                         /* just return current zone */
6071                         real_nzones = domi_nzones = 1;
6072                         zoneids = kmem_alloc(sizeof (zoneid_t), KM_SLEEP);
6073                         zoneids[0] = myzone->zone_id;
6074                 } else {
6075                         /* return all zones that are dominated */
6076                         mutex_enter(&zonehash_lock);
6077                         real_nzones = zonecount;
6078                         domi_nzones = 0;
6079                         if (real_nzones > 0) {
6080                                 zoneids = kmem_alloc(real_nzones *
6081                                     sizeof (zoneid_t), KM_SLEEP);
6082                                 mybslab = label2bslabel(myzone->zone_slabel);
6083                                 list_for_each(&zone_active, zone) {
6094                                 for (zone = list_head(&zone_active);
6095                                     zone != NULL;
6096                                     zone = list_next(&zone_active, zone)) {
6084                                         if (zone->zone_id == GLOBAL_ZONEID)
6085                                                 continue;
6086                                         if (zone != myzone &&
6087                                             (zone->zone_flags & ZF_IS_SCRATCH))
6088                                                 continue;
6089                                         /*
```



```
6090                                          * Note that a label always dominates
6091                                          * itself, so myzone is always included
6092                                          * in the list.
6093                                          */
6094                                         if (bldominates(mybslab,
6095                                             label2bslabel(zone->zone_slabel))) {
6096                                                 zoneids[domi_nzones++] =
6097                                                     zone->zone_id;
6098                                         }
6099                                 }
6100                         }
6101                         mutex_exit(&zonehash_lock);
6102                 }
6103         } else {
6104                 mutex_enter(&zonehash_lock);
6105                 real_nzones = zonecount;
6106                 domi_nzones = 0;
6107                 if (real_nzones > 0) {
6108                         zoneids = kmem_alloc(real_nzones * sizeof (zoneid_t),
6109                             KM_SLEEP);
6110                         list_for_each(&zone_active, zone)
6123                         for (zone = list_head(&zone_active); zone != NULL;
6124                             zone = list_next(&zone_active, zone))
6111                                 zoneids[domi_nzones++] = zone->zone_id;
6112                         ASSERT(domi_nzones == real_nzones);
6113                 }
6114                 mutex_exit(&zonehash_lock);
6115         }
6116
6117         /*
6118          * If user has allocated space for fewer entries than we found, then
6119          * return only up to his limit.  Either way, tell him exactly how many
6120          * we found.
6121          */
6122         if (domi_nzones < user_nzones)
6123                 user_nzones = domi_nzones;
6124         error = 0;
6125         if (copyout(&domi_nzones, numzones, sizeof (uint_t)) != 0) {
6126                 error = EFAULT;
6127         } else if (zoneidlist != NULL && user_nzones != 0) {
6128                 if (copyout(zoneids, zoneidlist,
6129                     user_nzones * sizeof (zoneid_t)) != 0)
6130                         error = EFAULT;
6131         }
6132
6133         if (real_nzones > 0)
6134                 kmem_free(zoneids, real_nzones * sizeof (zoneid_t));
6135
6136         if (error != 0)
6137                 return (set_errno(error));
6138         else
6139                 return (0);
6140 }
```

_____unchanged_portion_omitted_

```
6546 /*
6547  * Entry point so kadmin(A_SHUTDOWN, ...) can set the global zone's
6548  * status to ZONE_IS_SHUTTING_DOWN.
6549  *
6550  * This function also shuts down all running zones to ensure that they won't
6551  * fork new processes.
6552  */
6553 void
6554 zone_shutdown_global(void)
6555 {
6556         zone_t *current_zonep;
```

```
6558         ASSERT(INGLOBALZONE(curproc));
6559         mutex_enter(&zonehash_lock);
6560         mutex_enter(&zone_status_lock);

6562         /* Modify the global zone's status first. */
6563         ASSERT(zone_status_get(global_zone) == ZONE_IS_RUNNING);
6564         zone_status_set(global_zone, ZONE_IS_SHUTTING_DOWN);

6566         /*
6567          * Now change the states of all running zones to ZONE_IS_SHUTTING_DOWN.
6568          * We don't mark all zones with ZONE_IS_SHUTTING_DOWN because doing so
6569          * could cause assertions to fail (e.g., assertions about a zone's
6570          * state during initialization, readying, or booting) or produce races.
6571          * We'll let threads continue to initialize and ready new zones: they'll
6572          * fail to boot the new zones when they see that the global zone is
6573          * shutting down.
6574          */
6575         list_for_each(&zone_active, cpurrent_zonep) {
6589         for (current_zonep = list_head(&zone_active); current_zonep != NULL;
6590             current_zonep = list_next(&zone_active, current_zonep)) {
6576                 if (zone_status_get(current_zonep) == ZONE_IS_RUNNING)
6577                         zone_status_set(current_zonep, ZONE_IS_SHUTTING_DOWN);
6578         }
6579         mutex_exit(&zone_status_lock);
6580         mutex_exit(&zonehash_lock);
6581 }

6583 /*
6584  * Returns true if the named dataset is visible in the current zone.
6585  * The 'write' parameter is set to 1 if the dataset is also writable.
6586  */
6587 int
6588 zone_dataset_visible(const char *dataset, int *write)
6589 {
6590         static int zfstype = -1;
6591         zone_dataset_t *zd;
6592         size_t len;
6593         zone_t *zone = curproc->p_zone;
6594         const char *name = NULL;
6595         vfs_t *vfsp = NULL;

6597         if (dataset[0] == '\0')
6598                 return (0);

6600         /*
6601          * Walk the list once, looking for datasets which match exactly, or
6602          * specify a dataset underneath an exported dataset.  If found, return
6603          * true and note that it is writable.
6604          */
6605         list_for_each(&zone->zone_datasets, zd) {
6620         for (zd = list_head(&zone->zone_datasets); zd != NULL;
6621             zd = list_next(&zone->zone_datasets, zd)) {

6606                 len = strlen(zd->zd_dataset);
6607                 if (strlen(dataset) >= len &&
6608                     bcmp(dataset, zd->zd_dataset, len) == 0 &&
6609                     (dataset[len] == '\0' || dataset[len] == '/' ||
6610                     dataset[len] == '@')) {
6611                         if (write)
6612                                 *write = 1;
6613                         return (1);
6614                 }
6615         }

6617         /*
```



```
6618          * Walk the list a second time, searching for datasets which are parents
6619          * of exported datasets.  These should be visible, but read-only.
6620          *
6621          * Note that we also have to support forms such as 'pool/dataset/', with
6622          * a trailing slash.
6623          */
6624         list_for_each(&zone->zone_dataset, zd) {
6641         for (zd = list_head(&zone->zone_datasets); zd != NULL;
6642             zd = list_next(&zone->zone_datasets, zd)) {

6625                 len = strlen(dataset);
6626                 if (dataset[len - 1] == '/')
6627                         len--;  /* Ignore trailing slash */
6628                 if (len < strlen(zd->zd_dataset) &&
6629                     bcmp(dataset, zd->zd_dataset, len) == 0 &&
6630                     zd->zd_dataset[len] == '/') {
6631                         if (write)
6632                                 *write = 0;
6633                         return (1);
6634                 }
6635         }

6637         /*
6638          * We reach here if the given dataset is not found in the zone_dataset
6639          * list. Check if this dataset was added as a filesystem (ie. "add fs")
6640          * instead of delegation. For this we search for the dataset in the
6641          * zone_vfslist of this zone. If found, return true and note that it is
6642          * not writable.
6643          */

6645         /*
6646          * Initialize zfstype if it is not initialized yet.
6647          */
6648         if (zfstype == -1) {
6649                 struct vfssw *vswp = vfs_getvfssw("zfs");
6650                 zfstype = vswp - vfssw;
6651                 vfs_unrefvfssw(vswp);
6652         }

6654         vfs_list_read_lock();
6655         vfsp = zone->zone_vfslist;
6656         do {
6657                 ASSERT(vfsp);
6658                 if (vfsp->vfs_fstype == zfstype) {
6659                         name = refstr_value(vfsp->vfs_resource);

6661                         /*
6662                          * Check if we have an exact match.
6663                          */
6664                         if (strcmp(dataset, name) == 0) {
6665                                 vfs_list_unlock();
6666                                 if (write)
6667                                         *write = 0;
6668                                 return (1);
6669                         }
6670                         /*
6671                          * We need to check if we are looking for parents of
6672                          * a dataset. These should be visible, but read-only.
6673                          */
6674                         len = strlen(dataset);
6675                         if (dataset[len - 1] == '/')
6676                                 len--;

6678                         if (len < strlen(name) &&
6679                             bcmp(dataset, name, len) == 0 && name[len] == '/') {
6680                                 vfs_list_unlock();
```

```
6681                                if (write)
6682                                        *write = 0;
6683                                return (1);
6684                        }
6685                }
6686                vfsp = vfsp->vfs_zone_next;
6687        } while (vfsp != zone->zone_vfslist);

6689        vfs_list_unlock();
6690        return (0);
6691 }

6693 /*
6694  * zone_find_by_any_path() -
6695  *
6696  * kernel-private routine similar to zone_find_by_path(), but which
6697  * effectively compares against zone paths rather than zonerootpath
6698  * (i.e., the last component of zonerootpaths, which should be "root/",
6699  * are not compared.)  This is done in order to accurately identify all
6700  * paths, whether zone-visible or not, including those which are parallel
6701  * to /root/, such as /dev/, /home/, etc...
6702  *
6703  * If the specified path does not fall under any zone path then global
6704  * zone is returned.
6705  *
6706  * The treat_abs parameter indicates whether the path should be treated as
6707  * an absolute path although it does not begin with "/".  (This supports
6708  * nfs mount syntax such as host:any/path.)
6709  *
6710  * The caller is responsible for zone_rele of the returned zone.
6711  */
6712 zone_t *
6713 zone_find_by_any_path(const char *path, boolean_t treat_abs)
6714 {
6715        zone_t *zone;
6716        int path_offset = 0;

6718        if (path == NULL) {
6719                zone_hold(global_zone);
6720                return (global_zone);
6721        }

6723        if (*path != '/') {
6724                ASSERT(treat_abs);
6725                path_offset = 1;
6726        }

6728        mutex_enter(&zonehash_lock);
6729        list_for_each(&zone_active, zone) {
6748        for (zone = list_head(&zone_active); zone != NULL;
6749            zone = list_next(&zone_active, zone)) {
6730                char    *c;
6731                size_t  pathlen;
6732                char *rootpath_start;

6734                if (zone == global_zone)        /* skip global zone */
6735                        continue;

6737                /* scan backwards to find start of last component */
6738                c = zone->zone_rootpath + zone->zone_rootpathlen - 2;
6739                do {
6740                        c--;
6741                } while (*c != '/');

6743                pathlen = c - zone->zone_rootpath + 1 - path_offset;
6744                rootpath_start = (zone->zone_rootpath + path_offset);
```



```
6745                if (strncmp(path, rootpath_start, pathlen) == 0)
6746                        break;
6747        }
6748        if (zone == NULL)
6749                zone = global_zone;
6750        zone_hold(zone);
6751        mutex_exit(&zonehash_lock);
6752        return (zone);
6753 }

6755 /*
6756  * Finds a zone_dl_t with the given linkid in the given zone.  Returns the
6757  * zone_dl_t pointer if found, and NULL otherwise.
6758  */
6759 static zone_dl_t *
6760 zone_find_dl(zone_t *zone, datalink_id_t linkid)
6761 {
6762        zone_dl_t *zdl;

6764        ASSERT(mutex_owned(&zone->zone_lock));
6765        list_for_each(&zone->zone_dl_list, zdl) {
6785        for (zdl = list_head(&zone->zone_dl_list); zdl != NULL;
6786            zdl = list_next(&zone->zone_dl_list, zdl)) {
6766                if (zdl->zdl_id == linkid)
6767                        break;
6768        }
6769        return (zdl);
6770 }
______unchanged_portion_omitted_

6783 /*
6784  * Add an data link name for the zone.
6785  */
6786 static int
6787 zone_add_datalink(zoneid_t zoneid, datalink_id_t linkid)
6788 {
6789        zone_dl_t *zdl;
6790        zone_t *zone;
6791        zone_t *thiszone;

6793        if ((thiszone = zone_find_by_id(zoneid)) == NULL)
6794                return (set_errno(ENXIO));

6796        /* Verify that the datalink ID doesn't already belong to a zone. */
6797        mutex_enter(&zonehash_lock);
6798        list_for_each(&zone_active, zone) {
6819        for (zone = list_head(&zone_active); zone != NULL;
6820            zone = list_next(&zone_active, zone)) {
6799                if (zone_dl_exists(zone, linkid)) {
6800                        mutex_exit(&zonehash_lock);
6801                        zone_rele(thiszone);
6802                        return (set_errno((zone == thiszone) ? EEXIST : EPERM));
6803                }
6804        }

6806        zdl = kmem_zalloc(sizeof (*zdl), KM_SLEEP);
6807        zdl->zdl_id = linkid;
6808        zdl->zdl_net = NULL;
6809        mutex_enter(&thiszone->zone_lock);
6810        list_insert_head(&thiszone->zone_dl_list, zdl);
6811        mutex_exit(&thiszone->zone_lock);
6812        mutex_exit(&zonehash_lock);
6813        zone_rele(thiszone);
6814        return (0);
6815 }
______unchanged_portion_omitted_
```

```
/*
 * Using the zoneidp as ALL_ZONES, we can lookup which zone has been assigned
 * the linkid.  Otherwise we just check if the specified zoneidp has been
 * assigned the supplied linkid.
 */
int
zone_check_datalink(zoneid_t *zoneidp, datalink_id_t linkid)
{
        zone_t *zone;
        int err = ENXIO;

        if (*zoneidp != ALL_ZONES) {
                if ((zone = zone_find_by_id(*zoneidp)) != NULL) {
                        if (zone_dl_exists(zone, linkid))
                                err = 0;
                        zone_rele(zone);
                }
                return (err);
        }

        mutex_enter(&zonehash_lock);
        list_for_each(&zone_active, zone) {
        for (zone = list_head(&zone_active); zone != NULL;
            zone = list_next(&zone_active, zone)) {
                if (zone_dl_exists(zone, linkid)) {
                        *zoneidp = zone->zone_id;
                        err = 0;
                        break;
                }
        }
        mutex_exit(&zonehash_lock);
        return (err);
}

/*
 * Get the list of datalink IDs assigned to a zone.
 *
 * On input, *nump is the number of datalink IDs that can fit in the supplied
 * idarray.  Upon return, *nump is either set to the number of datalink IDs
 * that were placed in the array if the array was large enough, or to the
 * number of datalink IDs that the function needs to place in the array if the
 * array is too small.
 */
static int
zone_list_datalink(zoneid_t zoneid, int *nump, datalink_id_t *idarray)
{
        uint_t num, dlcount;
        zone_t *zone;
        zone_dl_t *zdl;
        datalink_id_t *idptr = idarray;

        if (copyin(nump, &dlcount, sizeof (dlcount)) != 0)
                return (set_errno(EFAULT));
        if ((zone = zone_find_by_id(zoneid)) == NULL)
                return (set_errno(ENXIO));

        num = 0;
        mutex_enter(&zone->zone_lock);
        list_for_each(&zone->zone_dl_list, zdl) {
        for (zdl = list_head(&zone->zone_dl_list); zdl != NULL;
            zdl = list_next(&zone->zone_dl_list, zdl)) {
                /*
                 * If the list is bigger than what the caller supplied, just
                 * count, don't do copyout.
                 */
```



```
                if (++num > dlcount)
                        continue;
                if (copyout(&zdl->zdl_id, idptr, sizeof (*idptr)) != 0) {
                        mutex_exit(&zone->zone_lock);
                        zone_rele(zone);
                        return (set_errno(EFAULT));
                }
                idptr++;
        }
        mutex_exit(&zone->zone_lock);
        zone_rele(zone);

        /* Increased or decreased, caller should be notified. */
        if (num != dlcount) {
                if (copyout(&num, nump, sizeof (num)) != 0)
                        return (set_errno(EFAULT));
        }
        return (0);
}
______unchanged_portion_omitted_

/*
 * Walk the datalinks for a given zone
 */
int
zone_datalink_walk(zoneid_t zoneid, int (*cb)(datalink_id_t, void *),
    void *data)
{
        zone_t          *zone;
        zone_dl_t       *zdl;
        datalink_id_t   *idarray;
        uint_t          idcount = 0;
        int             i, ret = 0;

        if ((zone = zone_find_by_id(zoneid)) == NULL)
                return (ENOENT);

        /*
         * We first build an array of linkid's so that we can walk these and
         * execute the callback with the zone_lock dropped.
         */
        mutex_enter(&zone->zone_lock);
        list_for_each(&zone->zone_dl_lists, zdl) {
        for (zdl = list_head(&zone->zone_dl_list); zdl != NULL;
            zdl = list_next(&zone->zone_dl_list, zdl)) {
                idcount++;
        }

        if (idcount == 0) {
                mutex_exit(&zone->zone_lock);
                zone_rele(zone);
                return (0);
        }

        idarray = kmem_alloc(sizeof (datalink_id_t) * idcount, KM_NOSLEEP);
        if (idarray == NULL) {
                mutex_exit(&zone->zone_lock);
                zone_rele(zone);
                return (ENOMEM);
        }

        i = 0;
        list_for_each(&zone->zone_dl_list, zdl) {
        for (i = 0, zdl = list_head(&zone->zone_dl_list); zdl != NULL;
            i++, zdl = list_next(&zone->zone_dl_list, zdl)) {
                idarray[i] = zdl->zdl_id;
```

```
6991                 i++;
6992 #endif /* ! codereview */
6993         }

6995         mutex_exit(&zone->zone_lock);

6997         for (i = 0; i < idcount && ret == 0; i++) {
6998                 if ((ret = (*cb)(idarray[i], data)) != 0)
6999                         break;
7000         }

7002         zone_rele(zone);
7003         kmem_free(idarray, sizeof (datalink_id_t) * idcount);
7004         return (ret);
7005 }

7007 static char *
7008 zone_net_type2name(int type)
7009 {
7010         switch (type) {
7011         case ZONE_NETWORK_ADDRESS:
7012                 return (ZONE_NET_ADDRNAME);
7013         case ZONE_NETWORK_DEFROUTER:
7014                 return (ZONE_NET_RTRNAME);
7015         default:
7016                 return (NULL);
7017         }
7018 }

7020 static int
7021 zone_set_network(zoneid_t zoneid, zone_net_data_t *znbuf)
7022 {
7023         zone_t *zone;
7024         zone_dl_t *zdl;
7025         nvlist_t *nvl;
7026         int err = 0;
7027         uint8_t *new = NULL;
7028         char *nvname;
7029         int bufsize;
7030         datalink_id_t linkid = znbuf->zn_linkid;

7032         if (secpolicy_zone_config(CRED()) != 0)
7033                 return (set_errno(EPERM));

7035         if (zoneid == GLOBAL_ZONEID)
7036                 return (set_errno(EINVAL));

7038         nvname = zone_net_type2name(znbuf->zn_type);
7039         bufsize = znbuf->zn_len;
7040         new = znbuf->zn_val;
7041         if (nvname == NULL)
7042                 return (set_errno(EINVAL));

7044         if ((zone = zone_find_by_id(zoneid)) == NULL) {
7045                 return (set_errno(EINVAL));
7046         }

7048         mutex_enter(&zone->zone_lock);
7049         if ((zdl = zone_find_dl(zone, linkid)) == NULL) {
7050                 err = ENXIO;
7051                 goto done;
7052         }
7053         if ((nvl = zdl->zdl_net) == NULL) {
7054                 if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP)) {
7055                         err = ENOMEM;
7056                         goto done;
```



```
7057                 } else {
7058                         zdl->zdl_net = nvl;
7059                 }
7060         }
7061         if (nvlist_exists(nvl, nvname)) {
7062                 err = EINVAL;
7063                 goto done;
7064         }
7065         err = nvlist_add_uint8_array(nvl, nvname, new, bufsize);
7066         ASSERT(err == 0);
7067 done:
7068         mutex_exit(&zone->zone_lock);
7069         zone_rele(zone);
7070         if (err != 0)
7071                 return (set_errno(err));
7072         else
7073                 return (0);
7074 }

7076 static int
7077 zone_get_network(zoneid_t zoneid, zone_net_data_t *znbuf)
7078 {
7079         zone_t *zone;
7080         zone_dl_t *zdl;
7081         nvlist_t *nvl;
7082         uint8_t *ptr;
7083         uint_t psize;
7084         int err = 0;
7085         char *nvname;
7086         int bufsize;
7087         void *buf;
7088         datalink_id_t linkid = znbuf->zn_linkid;

7090         if (zoneid == GLOBAL_ZONEID)
7091                 return (set_errno(EINVAL));

7093         nvname = zone_net_type2name(znbuf->zn_type);
7094         bufsize = znbuf->zn_len;
7095         buf = znbuf->zn_val;

7097         if (nvname == NULL)
7098                 return (set_errno(EINVAL));
7099         if ((zone = zone_find_by_id(zoneid)) == NULL)
7100                 return (set_errno(EINVAL));

7102         mutex_enter(&zone->zone_lock);
7103         if ((zdl = zone_find_dl(zone, linkid)) == NULL) {
7104                 err = ENXIO;
7105                 goto done;
7106         }
7107         if ((nvl = zdl->zdl_net) == NULL || !nvlist_exists(nvl, nvname)) {
7108                 err = ENOENT;
7109                 goto done;
7110         }
7111         err = nvlist_lookup_uint8_array(nvl, nvname, &ptr, &psize);
7112         ASSERT(err == 0);

7114         if (psize > bufsize) {
7115                 err = ENOBUFS;
7116                 goto done;
7117         }
7118         znbuf->zn_len = psize;
7119         bcopy(ptr, buf, psize);
7120 done:
7121         mutex_exit(&zone->zone_lock);
7122         zone_rele(zone);
```

```
7123         if (err != 0)
7124                 return (set_errno(err));
7125         else
7126                 return (0);
7127 }
```