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

```
/*
 * ZSD routines.
 *
 * Zone Specific Data (ZSD) is modeled after Thread Specific Data as
 * defined by the pthread_key_create() and related interfaces.
 *
 * Kernel subsystems may register one or more data items and/or
 * callbacks to be executed when a zone is created, shutdown, or
 * destroyed.
 *
 * Unlike the thread counterpart, destructor callbacks will be executed
 * even if the data pointer is NULL and/or there are no constructor
 * callbacks, so it is the responsibility of such callbacks to check for
 * NULL data values if necessary.
 *
 * The locking strategy and overall picture is as follows:
 *
 * When someone calls zone_key_create(), a template ZSD entry is added to the
 * global list "zsd_registered_keys", protected by zsd_key_lock.  While
 * holding that lock all the existing zones are marked as
 * ZSD_CREATE_NEEDED and a copy of the ZSD entry added to the per-zone
 * zone_zsd list (protected by zone_lock). The global list is updated first
 * (under zone_key_lock) to make sure that newly created zones use the
 * most recent list of keys. Then under zonehash_lock we walk the zones
 * and mark them.  Similar locking is used in zone_key_delete().
 *
 * The actual create, shutdown, and destroy callbacks are done without
 * holding any lock. And zsd_flags are used to ensure that the operations
 * completed so that when zone_key_create (and zone_create) is done, as well as
 * zone_key_delete (and zone_destroy) is done, all the necessary callbacks
 * are completed.
 *
 * When new zones are created constructor callbacks for all registered ZSD
 * entries will be called. That also uses the above two phases of marking
 * what needs to be done, and then running the callbacks without holding
 * any locks.
 *
 * The framework does not provide any locking around zone_getspecific() and
 * zone_setspecific() apart from that needed for internal consistency, so
 * callers interested in atomic "test-and-set" semantics will need to provide
 * their own locking.
 */

/*
 * Helper function to find the zsd_entry associated with the key in the
 * given list.
 */
static struct zsd_entry *
zsd_find(list_t *l, zone_key_t key)
{
        struct zsd_entry *zsd;

        list_for_each(l, zsd) {
        for (zsd = list_head(l); zsd != NULL; zsd = list_next(l, zsd)) {
                if (zsd->zsd_key == key) {
                        return (zsd);
                }
        }
        return (NULL);
```

```
}

/*
 * Helper function to find the zsd_entry associated with the key in the
 * given list. Move it to the front of the list.
 */
static struct zsd_entry *
zsd_find_mru(list_t *l, zone_key_t key)
{
        struct zsd_entry *zsd;

        list_for_each(l, zsd) {
        for (zsd = list_head(l); zsd != NULL; zsd = list_next(l, zsd)) {
                if (zsd->zsd_key == key) {
                        /*
                         * Move to head of list to keep list in MRU order.
                         */
                        if (zsd != list_head(l)) {
                                list_remove(l, zsd);
                                list_insert_head(l, zsd);
                        }
                        return (zsd);
                }
        }
        return (NULL);
}

void
zone_key_create(zone_key_t *keyp, void *(*create)(zoneid_t),
    void (*shutdown)(zoneid_t, void *), void (*destroy)(zoneid_t, void *))
{
        struct zsd_entry *zsdp;
        struct zsd_entry *t;
        struct zone *zone;
        zone_key_t  key;

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

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

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

                mutex_enter(&zone->zone_lock);

                /* Skip zones that are on the way down or not yet up */
                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
 * the template list (zsd_registered_keys). The constructor callback is
 * executed later (once the zone exists and with locks dropped).
 */
static void
zone_zsd_configure(zone_t *zone)
{
        struct zsd_entry *zsdp;
        struct zsd_entry *t;

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

                t = kmem_zalloc(sizeof (*t), KM_SLEEP);
                t->zsd_key = zsdp->zsd_key;
                t->zsd_create = zsdp->zsd_create;
                t->zsd_shutdown = zsdp->zsd_shutdown;
                t->zsd_destroy = zsdp->zsd_destroy;
                if (zsdp->zsd_create != NULL) {
                        t->zsd_flags = ZSD_CREATE_NEEDED;
                        DTRACE_PROBE2(zsd__create__needed,
                            zone_t *, zone, zone_key_t, zsdp->zsd_key);
                }
                list_insert_tail(&zone->zone_zsd, t);
        }
        mutex_exit(&zsd_key_lock);
        mutex_exit(&zone->zone_lock);
}

enum zsd_callback_type { ZSD_CREATE, ZSD_SHUTDOWN, ZSD_DESTROY };

/*
 * Helper function to execute shutdown or destructor callbacks.
 */
static void
zone_zsd_callbacks(zone_t *zone, enum zsd_callback_type ct)
{
        struct zsd_entry *t;

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

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

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

```
                if (ct == ZSD_SHUTDOWN) {
                        if (t->zsd_shutdown != NULL &&
                            (t->zsd_flags & ZSD_SHUTDOWN_ALL) == 0) {
                                t->zsd_flags |= ZSD_SHUTDOWN_NEEDED;
                                DTRACE_PROBE2(zsd__shutdown__needed,
                                    zone_t *, zone, zone_key_t, key);
                        }
                } else {
                        if (t->zsd_destroy != NULL &&
                            (t->zsd_flags & ZSD_DESTROY_ALL) == 0) {
                                t->zsd_flags |= ZSD_DESTROY_NEEDED;
                                DTRACE_PROBE2(zsd__destroy__needed,
                                    zone_t *, zone, zone_key_t, key);
                        }
                }
        }
        mutex_exit(&zone->zone_lock);

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

}

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

        /*
         * Free all the zsd_entry's we had on this zone.
         */
        mutex_enter(&zone->zone_lock);
        list_for_each_safe(&zone->zone_zsd, t, next) {
        for (t = list_head(&zone->zone_zsd); t != NULL; t = next) {
                next = list_next(&zone->zone_zsd, t);
                list_remove(&zone->zone_zsd, t);
                ASSERT(!(t->zsd_flags & ZSD_ALL_INPROGRESS));
                kmem_free(t, sizeof (*t));
        }
        list_destroy(&zone->zone_zsd);
        mutex_exit(&zone->zone_lock);

}
```

______***unchanged_portion_omitted_***

```
/*
 * Frees memory associated with the zone dataset list.
 */
static void
zone_free_datasets(zone_t *zone)
{
        zone_dataset_t *t, *next;

        list_for_each_safe(&zone->zone_datasets, t, next) {
        for (t = list_head(&zone->zone_datasets); t != NULL; t = next) {
                next = list_next(&zone->zone_datasets, t);
                list_remove(&zone->zone_datasets, t);
                kmem_free(t->zd_dataset, strlen(t->zd_dataset) + 1);
                kmem_free(t, sizeof (*t));
        }
        list_destroy(&zone->zone_datasets);
```

```
}
______unchanged_portion_omitted_

/*
 * Similar to zone_find_by_id(), using the path as a key.  For instance,
 * if there is a zone "foo" rooted at /foo/root, and the path argument
 * is "/foo/root/proc", it will return the held zone_t corresponding to
 * zone "foo".
 *
 * zone_find_by_path() always returns a non-NULL value, since at the
 * very least every path will be contained in the global zone.
 *
 * As with the other zone_find_by_*() functions, the caller is
 * responsible for zone_rele()ing the return value of this function.
 */
zone_t *
zone_find_by_path(const char *path)
{
        zone_t *zone;
        zone_t *zret = NULL;
        zone_status_t status;

        if (path == NULL) {
                /*
                 * Call from rootconf().
                 */
                zone_hold(global_zone);
                return (global_zone);
        }
        ASSERT(*path == '/');
        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_PATH_VISIBLE(path, zone))
                        zret = zone;
        }
        ASSERT(zret != NULL);
        status = zone_status_get(zret);
        if (status < ZONE_IS_READY || status > ZONE_IS_DOWN) {
                /*
                 * Zone practically doesn't exist.
                 */
                zret = global_zone;
        }
        zone_hold(zret);
        mutex_exit(&zonehash_lock);
        return (zret);
}
______unchanged_portion_omitted_

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

```
        mutex_enter(&zonehash_lock);
        list_for_each(&zone_active, zone) {
        for (zone = list_head(&zone_active); zone != NULL;
            zone = list_next(&zone_active, zone)) {
                /*
                 * Skip zones that shouldn't be externally visible.
                 */
                status = zone_status_get(zone);
                if (status < ZONE_IS_READY || status > ZONE_IS_DOWN)
                        continue;
                /*
                 * Bail immediately if any callback invocation returns a
                 * non-zero value.
                 */
                ret = (*cb)(zone, data);
                if (ret != 0)
                        break;
        }
        mutex_exit(&zonehash_lock);
        return (ret);
}
______unchanged_portion_omitted_

/*
 * Helper function to make sure that a zone created on 'rootpath'
 * wouldn't end up containing other zones' rootpaths.
 */
static boolean_t
zone_is_nested(const char *rootpath)
{
        zone_t *zone;
        size_t rootpathlen = strlen(rootpath);
        size_t len;

        ASSERT(MUTEX_HELD(&zonehash_lock));

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

        list_for_each(&zone_active, zone) {
        for (zone = list_head(&zone_active); zone != NULL;
            zone = list_next(&zone_active, zone)) {
                if (zone == global_zone)
                        continue;
                len = strlen(zone->zone_rootpath);
                if (strncmp(rootpath, zone->zone_rootpath,
                    MIN(rootpathlen, len)) == 0)
                        return (B_TRUE);
        }
        return (B_FALSE);
}
______unchanged_portion_omitted_

/*
 * Systemcall entry point for zone_enter().
 *
 * The current process is injected into said zone.  In the process
 * it will change its project membership, privileges, rootdir/cwd,
 * zone-wide rctls, and pool association to match those of the zone.
 *
 * The first zone_enter() called while the zone is in the ZONE_IS_READY
 * state will transition it to ZONE_IS_RUNNING.  Processes may only
 * enter a zone that is "ready" or "running".
 */
static int
zone_enter(zoneid_t zoneid)
{
        zone_t *zone;
        vnode_t *vp;
        proc_t *pp = curproc;
        contract_t *ct;
        cont_process_t *ctp;
        task_t *tk, *oldtk;
        kproject_t *zone_proj0;
        cred_t *cr, *newcr;
        pool_t *oldpool, *newpool;
        sess_t *sp;
        uid_t uid;
        zone_status_t status;
        int err = 0;
        rctl_entity_p_t e;
        size_t swap;
        kthread_id_t t;

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

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

        /*
         * Make sure we're not changing zones with files open or mapped in
         * to our address space which shouldn't be changing zones.
         */
        if (!files_can_change_zones()) {
                err = EBADF;
                goto out;
        }
        if (!as_can_change_zones()) {
                err = EFAULT;
                goto out;
        }

        mutex_enter(&zonehash_lock);
        if (pp->p_zone != global_zone) {
                mutex_exit(&zonehash_lock);
                err = EINVAL;
                goto out;
        }

        zone = zone_find_all_by_id(zoneid);
        if (zone == NULL) {
                mutex_exit(&zonehash_lock);
                err = EINVAL;
                goto out;
        }

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

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

        /*
         * Moreover, we don't allow processes whose encapsulating
         * process contracts have inherited extrazonal contracts.
         * While it would be easier to eliminate all process contracts
         * with inherited contracts, we need to be able to give a
         * restarted init (or other zone-penetrating process) its
         * predecessor's contracts.
         */
        if (ctp->conp_ninherited != 0) {
                contract_t *next;
                list_for_each(&ctp->conp_inherited, next) {
                for (next = list_head(&ctp->conp_inherited); next;
                    next = list_next(&ctp->conp_inherited, next)) {
                        if (contract_getzuniqid(next) != zone->zone_uniqid) {
                                mutex_exit(&pp->p_lock);
                                mutex_exit(&ct->ct_lock);
                                mutex_exit(&zonehash_lock);
                                err = EINVAL;
                                goto out;
                        }
                }
        }

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

        status = zone_status_get(zone);
        if (status < ZONE_IS_READY || status >= ZONE_IS_SHUTTING_DOWN) {
                /*
                 * Can't join
                 */
                mutex_exit(&zonehash_lock);
                err = EINVAL;
                goto out;
        }

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

        /*
         * Grab pool_lock to keep the pools configuration from changing
         * and to stop ourselves from getting rebound to another pool
         * until we join the zone.
         */
        if (pool_lock_intr() != 0) {
                zone_rele(zone);
                err = EINTR;
                goto out;
        }
        ASSERT(secpolicy_pool(CRED()) == 0);
        /*
         * Bind ourselves to the pool currently associated with the zone.
         */
        oldpool = curproc->p_pool;
        newpool = zone_pool_get(zone);
        if (pool_state == POOL_ENABLED && newpool != oldpool &&
            (err = pool_do_bind(newpool, P_PID, P_MYID,
            POOL_BIND_ALL)) != 0) {
                pool_unlock();
                zone_rele(zone);
                goto out;
        }

        /*
         * Grab cpu_lock now; we’ll need it later when we call
         * task_join().
         */
        mutex_enter(&cpu_lock);
        mutex_enter(&zonehash_lock);
        /*
         * Make sure the zone hasn’t moved on since we dropped zonehash_lock.
         */
        if (zone_status_get(zone) >= ZONE_IS_SHUTTING_DOWN) {
                /*
                 * Can’t join anymore.
                 */
                mutex_exit(&zonehash_lock);
                mutex_exit(&cpu_lock);
                if (pool_state == POOL_ENABLED &&
                    newpool != oldpool)
                        (void) pool_do_bind(oldpool, P_PID, P_MYID,
                            POOL_BIND_ALL);
                pool_unlock();
                zone_rele(zone);
                err = EINVAL;
                goto out;
        }

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

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

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

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

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

        /* remove lwps and process from proc’s old zone and old project */
        mutex_enter(&pp->p_zone->zone_nlwps_lock);
        pp->p_zone->zone_nlwps -= pp->p_lwpcnt;
        pp->p_task->tk_proj->kpj_nlwps -= pp->p_lwpcnt;
        pp->p_task->tk_proj->kpj_nprocs--;
        pp->p_zone->zone_nprocs--;
        mutex_exit(&pp->p_zone->zone_nlwps_lock);

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

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

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

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

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

        /*
         * Reset the encapsulating process contract’s zone.
         */
        ASSERT(ct->ct_mzuniqid == GLOBAL_ZONEUNIQID);
        contract_setzuniqid(ct, zone->zone_uniqid);

        /*
```

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

        /*
         * call RCTLOP_SET functions on this proc
         */
        e.rcep_p.zone = zone;
        e.rcep_t = RCENTITY_ZONE;
        (void) rctl_set_dup(NULL, NULL, pp, &e, zone->zone_rctls, NULL,
            RCD_CALLBACK);
        mutex_exit(&pp->p_lock);

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

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

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

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

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

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

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

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

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

        mutex_exit(&zonehash_lock);
        /*
         * We're firmly in the zone; let pools progress.
         */
        pool_unlock();
        task_rele(oldtk);
        /*
         * We don't need to retain a hold on the zone since we already
         * incremented zone_ntasks, so the zone isn't going anywhere.
         */
        zone_rele(zone);

        /*
         * Chroot
         */
        vp = zone->zone_rootvp;
        zone_chdir(vp, &PTOU(pp)->u_cdir, pp);
        zone_chdir(vp, &PTOU(pp)->u_rdir, pp);

        /*
         * Change process credentials
         */
        newcr = cralloc();
        mutex_enter(&pp->p_crlock);
        cr = pp->p_cred;
        crcopy_to(cr, newcr);
        crsetzone(newcr, zone);
        pp->p_cred = newcr;

        /*
         * Restrict all process privilege sets to zone limit
         */
        priv_intersect(zone->zone_privset, &CR_PPRIV(newcr));
        priv_intersect(zone->zone_privset, &CR_EPRIV(newcr));
        priv_intersect(zone->zone_privset, &CR_IPRIV(newcr));
        priv_intersect(zone->zone_privset, &CR_LPRIV(newcr));
        mutex_exit(&pp->p_crlock);
        crset(pp, newcr);

        /*
         * Adjust upcount to reflect zone entry.
         */
        uid = crgetruid(newcr);
        mutex_enter(&pidlock);
        upcount_dec(uid, GLOBAL_ZONEID);
        upcount_inc(uid, zoneid);
        mutex_exit(&pidlock);

        /*
         * Set up core file path and content.
         */
        set_core_defaults();

out:
        /*
         * Let the other lwps continue.
         */
        mutex_enter(&pp->p_lock);
        if (curthread != pp->p_agenttp)
                continuelwps(pp);
        mutex_exit(&pp->p_lock);

        return (err != 0 ? set_errno(err) : 0);
}

/*
 * Systemcall entry point for zone_list(2).
 *
 * Processes running in a (non-global) zone only see themselves.
 * On labeled systems, they see all zones whose label they dominate.
 */
static int
zone_list(zoneid_t *zoneidlist, uint_t *numzones)
{
        zoneid_t *zoneids;
        zone_t *zone, *myzone;
        uint_t user_nzones, real_nzones;
        uint_t domi_nzones;
        int error;

        if (copyin(numzones, &user_nzones, sizeof (uint_t)) != 0)
                return (set_errno(EFAULT));

        myzone = curproc->p_zone;
        if (myzone != global_zone) {
                bslabel_t *mybslab;

                if (!is_system_labeled()) {
                        /* just return current zone */
                        real_nzones = domi_nzones = 1;
                        zoneids = kmem_alloc(sizeof (zoneid_t), KM_SLEEP);
                        zoneids[0] = myzone->zone_id;
                } else {
                        /* return all zones that are dominated */
                        mutex_enter(&zonehash_lock);
                        real_nzones = zonecount;
                        domi_nzones = 0;
                        if (real_nzones > 0) {
                                zoneids = kmem_alloc(real_nzones *
                                    sizeof (zoneid_t), KM_SLEEP);
                                mybslab = label2bslabel(myzone->zone_slabel);
                                list_for_each(&zone_active, zone) {
                                for (zone = list_head(&zone_active);
                                    zone != NULL;
                                    zone = list_next(&zone_active, zone)) {
                                        if (zone->zone_id == GLOBAL_ZONEID)
                                                continue;
                                        if (zone != myzone &&
                                            (zone->zone_flags & ZF_IS_SCRATCH))
                                                continue;
                                        /*
```

```
                                         * Note that a label always dominates
                                         * itself, so myzone is always included
                                         * in the list.
                                         */
                                        if (bldominates(mybslab,
                                            label2bslabel(zone->zone_slabel))) {
                                                zoneids[domi_nzones++] =
                                                    zone->zone_id;
                                        }
                                }
                        }
                        mutex_exit(&zonehash_lock);
                }
        } else {
                mutex_enter(&zonehash_lock);
                real_nzones = zonecount;
                domi_nzones = 0;
                if (real_nzones > 0) {
                        zoneids = kmem_alloc(real_nzones * sizeof (zoneid_t),
                            KM_SLEEP);
                        list_for_each(&zone_active, zone)
                        for (zone = list_head(&zone_active); zone != NULL;
                            zone = list_next(&zone_active, zone))
                                zoneids[domi_nzones++] = zone->zone_id;
                        ASSERT(domi_nzones == real_nzones);
                }
                mutex_exit(&zonehash_lock);
        }

        /*
         * If user has allocated space for fewer entries than we found, then
         * return only up to his limit.  Either way, tell him exactly how many
         * we found.
         */
        if (domi_nzones < user_nzones)
                user_nzones = domi_nzones;
        error = 0;
        if (copyout(&domi_nzones, numzones, sizeof (uint_t)) != 0) {
                error = EFAULT;
        } else if (zoneidlist != NULL && user_nzones != 0) {
                if (copyout(zoneids, zoneidlist,
                    user_nzones * sizeof (zoneid_t)) != 0)
                        error = EFAULT;
        }

        if (real_nzones > 0)
                kmem_free(zoneids, real_nzones * sizeof (zoneid_t));

        if (error != 0)
                return (set_errno(error));
        else
                return (0);
}
______unchanged_portion_omitted_

/*
 * Entry point so kadmin(A_SHUTDOWN, ...) can set the global zone’s
 * status to ZONE_IS_SHUTTING_DOWN.
 *
 * This function also shuts down all running zones to ensure that they won’t
 * fork new processes.
 */
void
zone_shutdown_global(void)
{
        zone_t *current_zonep;
        ASSERT(INGLOBALZONE(curproc));
        mutex_enter(&zonehash_lock);
        mutex_enter(&zone_status_lock);

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

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

/*
 * Returns true if the named dataset is visible in the current zone.
 * The 'write' parameter is set to 1 if the dataset is also writable.
 */
int
zone_dataset_visible(const char *dataset, int *write)
{
        static int zfstype = -1;
        zone_dataset_t *zd;
        size_t len;
        zone_t *zone = curproc->p_zone;
        const char *name = NULL;
        vfs_t *vfsp = NULL;

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

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

                len = strlen(zd->zd_dataset);
                if (strlen(dataset) >= len &&
                    bcmp(dataset, zd->zd_dataset, len) == 0 &&
                    (dataset[len] == '\0' || dataset[len] == '/' ||
                    dataset[len] == '@')) {
                        if (write)
                                *write = 1;
                        return (1);
                }
        }

        /*
```

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

                len = strlen(dataset);
                if (dataset[len - 1] == '/')
                        len--;  /* Ignore trailing slash */
                if (len < strlen(zd->zd_dataset) &&
                    bcmp(dataset, zd->zd_dataset, len) == 0 &&
                    zd->zd_dataset[len] == '/') {
                        if (write)
                                *write = 0;
                        return (1);
                }
        }

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

        /*
         * Initialize zfstype if it is not initialized yet.
         */
        if (zfstype == -1) {
                struct vfssw *vswp = vfs_getvfssw("zfs");
                zfstype = vswp - vfssw;
                vfs_unrefvfssw(vswp);
        }

        vfs_list_read_lock();
        vfsp = zone->zone_vfslist;
        do {
                ASSERT(vfsp);
                if (vfsp->vfs_fstype == zfstype) {
                        name = refstr_value(vfsp->vfs_resource);

                        /*
                         * Check if we have an exact match.
                         */
                        if (strcmp(dataset, name) == 0) {
                                vfs_list_unlock();
                                if (write)
                                        *write = 0;
                                return (1);
                        }
                        /*
                         * We need to check if we are looking for parents of
                         * a dataset. These should be visible, but read-only.
                         */
                        len = strlen(dataset);
                        if (dataset[len - 1] == '/')
                                len--;

                        if (len < strlen(name) &&
                            bcmp(dataset, name, len) == 0 && name[len] == '/') {
                                vfs_list_unlock();
                                if (write)
                                        *write = 0;
                                return (1);
                        }
                }
                vfsp = vfsp->vfs_zone_next;
        } while (vfsp != zone->zone_vfslist);

        vfs_list_unlock();
        return (0);
}

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

        if (path == NULL) {
                zone_hold(global_zone);
                return (global_zone);
        }

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

        mutex_enter(&zonehash_lock);
        list_for_each(&zone_active, zone) {
        for (zone = list_head(&zone_active); zone != NULL;
            zone = list_next(&zone_active, zone)) {
                char    *c;
                size_t  pathlen;
                char *rootpath_start;

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

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

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

```
                if (strncmp(path, rootpath_start, pathlen) == 0)
                        break;
        }
        if (zone == NULL)
                zone = global_zone;
        zone_hold(zone);
        mutex_exit(&zonehash_lock);
        return (zone);
}

/*
 * Finds a zone_dl_t with the given linkid in the given zone.  Returns the
 * zone_dl_t pointer if found, and NULL otherwise.
 */
static zone_dl_t *
zone_find_dl(zone_t *zone, datalink_id_t linkid)
{
        zone_dl_t *zdl;

        ASSERT(mutex_owned(&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 (zdl->zdl_id == linkid)
                        break;
        }
        return (zdl);
}
______unchanged_portion_omitted_

/*
 * Add an data link name for the zone.
 */
static int
zone_add_datalink(zoneid_t zoneid, datalink_id_t linkid)
{
        zone_dl_t *zdl;
        zone_t *zone;
        zone_t *thiszone;

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

        /* Verify that the datalink ID doesn't already belong to a 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)) {
                if (zone_dl_exists(zone, linkid)) {
                        mutex_exit(&zonehash_lock);
                        zone_rele(thiszone);
                        return (set_errno((zone == thiszone) ? EEXIST : EPERM));
                }
        }

        zdl = kmem_zalloc(sizeof (*zdl), KM_SLEEP);
        zdl->zdl_id = linkid;
        zdl->zdl_net = NULL;
        mutex_enter(&thiszone->zone_lock);
        list_insert_head(&thiszone->zone_dl_list, zdl);
        mutex_exit(&thiszone->zone_lock);
        mutex_exit(&zonehash_lock);
        zone_rele(thiszone);
        return (0);
}
______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;
                i++;
#endif /* ! codereview */
        }

        mutex_exit(&zone->zone_lock);

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

        zone_rele(zone);
        kmem_free(idarray, sizeof (datalink_id_t) * idcount);
        return (ret);
}

static char *
zone_net_type2name(int type)
{
        switch (type) {
        case ZONE_NETWORK_ADDRESS:
                return (ZONE_NET_ADDRNAME);
        case ZONE_NETWORK_DEFROUTER:
                return (ZONE_NET_RTRNAME);
        default:
                return (NULL);
        }
}

static int
zone_set_network(zoneid_t zoneid, zone_net_data_t *znbuf)
{
        zone_t *zone;
        zone_dl_t *zdl;
        nvlist_t *nvl;
        int err = 0;
        uint8_t *new = NULL;
        char *nvname;
        int bufsize;
        datalink_id_t linkid = znbuf->zn_linkid;

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

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

        nvname = zone_net_type2name(znbuf->zn_type);
        bufsize = znbuf->zn_len;
        new = znbuf->zn_val;
        if (nvname == NULL)
                return (set_errno(EINVAL));

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

        mutex_enter(&zone->zone_lock);
        if ((zdl = zone_find_dl(zone, linkid)) == NULL) {
                err = ENXIO;
                goto done;
        }
        if ((nvl = zdl->zdl_net) == NULL) {
                if (nvlist_alloc(&nvl, NV_UNIQUE_NAME, KM_SLEEP)) {
                        err = ENOMEM;
                        goto done;
```

```
                } else {
                        zdl->zdl_net = nvl;
                }
        }
        if (nvlist_exists(nvl, nvname)) {
                err = EINVAL;
                goto done;
        }
        err = nvlist_add_uint8_array(nvl, nvname, new, bufsize);
        ASSERT(err == 0);
done:
        mutex_exit(&zone->zone_lock);
        zone_rele(zone);
        if (err != 0)
                return (set_errno(err));
        else
                return (0);
}

static int
zone_get_network(zoneid_t zoneid, zone_net_data_t *znbuf)
{
        zone_t *zone;
        zone_dl_t *zdl;
        nvlist_t *nvl;
        uint8_t *ptr;
        uint_t psize;
        int err = 0;
        char *nvname;
        int bufsize;
        void *buf;
        datalink_id_t linkid = znbuf->zn_linkid;

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

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

        if (nvname == NULL)
                return (set_errno(EINVAL));
        if ((zone = zone_find_by_id(zoneid)) == NULL)
                return (set_errno(EINVAL));

        mutex_enter(&zone->zone_lock);
        if ((zdl = zone_find_dl(zone, linkid)) == NULL) {
                err = ENXIO;
                goto done;
        }
        if ((nvl = zdl->zdl_net) == NULL || !nvlist_exists(nvl, nvname)) {
                err = ENOENT;
                goto done;
        }
        err = nvlist_lookup_uint8_array(nvl, nvname, &ptr, &psize);
        ASSERT(err == 0);

        if (psize > bufsize) {
                err = ENOBUFS;
                goto done;
        }
        znbuf->zn_len = psize;
        bcopy(ptr, buf, psize);
done:
        mutex_exit(&zone->zone_lock);
        zone_rele(zone);
        if (err != 0)
                return (set_errno(err));
        else
                return (0);
}
```