Subject: [PATCH 08/10] Containers(V10): Share css_group arrays between tasks with same container memberships
Posted by Paul Menage on Tue, 29 May 2007 13:01:12 GMT
View Forum Message <> Reply to Message

This patch replaces the struct css_group embedded in task_struct with
a pointer; all tasks that have the same set of memberships across all
hierarchies will share a css_group object, and will be linked via
their css_groups field to the "tasks" list_head in the css_group.

Assuming that many tasks share the same container assignments, this
reduces overall space usage and keeps the size of the task_struct down
(three pointers added to task_struct compared to a non-containers
kernel, no matter how many subsystems are registered).

Signed-off-by: Paul Menage <menage@google.com>

 Documentation/containers.txt | 14 +
 include/linux/container.h | 93 ++++++-
 include/linux/sched.h | 33 --
 kernel/container.c | 524 +++++++++++++++++++++++++++++++++++++------
 kernel/cpuset.c | 15 -
 5 files changed, 553 insertions(+), 126 deletions(-)

Index: container-2.6.22-rc2-mm1/include/linux/container.h
 ========== =======
--- container-2.6.22-rc2-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc2-mm1/include/linux/container.h
@@ -29,6 +29,14 @@ extern void container_unlock(void);

 struct containerfs_root;

+/* Define the enumeration of all container subsystems */
+#define SUBSYS(_x) _x ## _subsys_id,
+enum container_subsys_id {
+#include <linux/container_subsys.h>
+ CONTAINER_SUBSYS_COUNT
+};
+#undef SUBSYS
+
 /* Per-subsystem/per-container state maintained by the system. */
 struct container_subsys_state {
  /* The container that this subsystem is attached to. Useful
@@ -85,6 +93,54 @@ struct container {

  struct containerfs_root *root;
  struct container *top_container;

```
+
+ /*
+  * List of cg_container_links pointing at css_groups with
+  * tasks in this container. Protected by css_group_lock
+  */
+ struct list_head css_groups;
+};
+
+/* A css_group is a structure holding pointers to a set of
+ * container_subsys_state objects. This saves space in the task struct
+ * object and speeds up fork()/exit(), since a single inc/dec and a
+ * list_add()/del() can bump the reference count on the entire
+ * container set for a task.
+ */
+
+struct css_group {
+
+ /* Reference count */
+ struct kref ref;
+
+ /*
+  * List running through all container groups. Protected by
+  * css_group_lock
+  */
+ struct list_head list;
+
+ /*
+  * List running through all tasks using this container
+  * group. Protected by css_group_lock
+  */
+ struct list_head tasks;
+
+ /*
+  * List of cg_container_link objects on link chains from
+  * containers referenced from this css_group. Protected by
+  * css_group_lock
+  */
+ struct list_head cg_links;
+
+ /* Set of subsystem states, one for each subsystem. NULL for
+  * subsystems that aren't part of this hierarchy. These
+  * pointers reduce the number of dereferences required to get
+  * from a task to its state for a given container, but result
+  * in increased space usage if tasks are in wildly different
+  * groupings across different hierarchies. This array is
+  * immutable after creation */
+ struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
+
```

```
 };

 /* struct cftype:
@@ -111,6 +167,10 @@ struct cftype {
  ssize_t (*read) (struct container *cont, struct cftype *cft,
     struct file *file,
     char __user *buf, size_t nbytes, loff_t *ppos);
+ /*
+  * read_uint() is a shortcut for the common case of returning a
+  * single integer. Use it in place of read()
+  */
  u64 (*read_uint) (struct container *cont, struct cftype *cft);
  ssize_t (*write) (struct container *cont, struct cftype *cft,
      struct file *file,
@@ -131,15 +191,7 @@ int container_is_removed(const struct co

 int container_path(const struct container *cont, char *buf, int buflen);

-int __container_task_count(const struct container *cont);
-static inline int container_task_count(const struct container *cont)
-{
- int task_count;
- rcu_read_lock();
- task_count = __container_task_count(cont);
- rcu_read_unlock();
- return task_count;
-}
+int container_task_count(const struct container *cont);

 /* Return true if the container is a descendant of the current container */
 int container_is_descendant(const struct container *cont);
@@ -186,7 +238,7 @@ static inline struct container_subsys_st
 static inline struct container_subsys_state *task_subsys_state(
  struct task_struct *task, int subsys_id)
 {
- return rcu_dereference(task->containers.subsys[subsys_id]);
+ return rcu_dereference(task->containers->subsys[subsys_id]);
 }

 static inline struct container* task_container(struct task_struct *task,
@@ -199,6 +251,27 @@ int container_path(const struct containe

 int container_clone(struct task_struct *tsk, struct container_subsys *ss);

+/* A container_iter should be treated as an opaque object */
+struct container_iter {
+ struct list_head *cg_link;
+ struct list_head *task;
```

```
+};
+
+/* To iterate across the tasks in a container:
+ *
+ * 1) call container_iter_start to intialize an iterator
+ *
+ * 2) call container_iter_next() to retrieve member tasks until it
+ *    returns NULL or until you want to end the iteration
+ *
+ * 3) call container_iter_end() to destroy the iterator.
+ */
+void container_iter_start(struct container *cont, struct container_iter *it);
+struct task_struct *container_iter_next(struct container *cont,
+     struct container_iter *it);
+void container_iter_end(struct container *cont, struct container_iter *it);
+
+
 #else /* !CONFIG_CONTAINERS */

 static inline int container_init_early(void) { return 0; }
Index: container-2.6.22-rc2-mm1/include/linux/sched.h
 ========================================================== =======
--- container-2.6.22-rc2-mm1.orig/include/linux/sched.h
+++ container-2.6.22-rc2-mm1/include/linux/sched.h
@@ -849,34 +849,6 @@ struct sched_class {
  void (*task_new) (struct rq *rq, struct task_struct *p);
 };

-#ifdef CONFIG_CONTAINERS
-
-#define SUBSYS(_x) _x ## _subsys_id,
-enum container_subsys_id {
-#include <linux/container_subsys.h>
- CONTAINER_SUBSYS_COUNT
-};
-#undef SUBSYS
-
-/* A css_group is a structure holding pointers to a set of
- * container_subsys_state objects.
- */
-
-struct css_group {
-
- /* Set of subsystem states, one for each subsystem. NULL for
-  * subsystems that aren't part of this hierarchy. These
-  * pointers reduce the number of dereferences required to get
-  * from a task to its state for a given container, but result
-  * in increased space usage if tasks are in wildly different
```

```
-  * groupings across different hierarchies. This array is
-  * immutable after creation */
- struct container_subsys_state *subsys[CONTAINER_SUBSYS_COUNT];
-
-};
-
-#endif /* CONFIG_CONTAINERS */
-
 struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
@@ -1133,7 +1105,10 @@ struct task_struct {
  int cpuset_mem_spread_rotor;
 #endif
 #ifdef CONFIG_CONTAINERS
- struct css_group containers;
+ /* Container info protected by css_group_lock */
+ struct css_group *containers;
+ /* cg_list protected by css_group_lock and tsk->alloc_lock */
+ struct list_head cg_list;
 #endif
  struct robust_list_head __user *robust_list;
 #ifdef CONFIG_COMPAT
Index: container-2.6.22-rc2-mm1/kernel/container.c
 ===================================================================== =======
--- container-2.6.22-rc2-mm1.orig/kernel/container.c
+++ container-2.6.22-rc2-mm1/kernel/container.c
@@ -101,6 +101,7 @@ static struct containerfs_root rootnode;
 /* The list of hierarchy roots */

 static LIST_HEAD(roots);
+static int root_count;

 /* dummytop is a shorthand for the dummy hierarchy's top container */
 #define dummytop (&rootnode.top_container)
@@ -132,12 +133,43 @@ list_for_each_entry(_ss, &_root->subsys_
 #define for_each_root(_root) \
 list_for_each_entry(_root, &roots, root_list)

-/* Each task_struct has an embedded css_group, so the get/put
- * operation simply takes a reference count on all the containers
- * referenced by subsystems in this css_group. This can end up
- * multiple-counting some containers, but that's OK - the ref-count is
- * just a busy/not-busy indicator; ensuring that we only count each
- * container once would require taking a global lock to ensure that no
+/* Link structure for associating css_group objects with containers */
+struct cg_container_link {
+ /*
```

```
+  * List running through cg_container_links associated with a
+  * container, anchored on container->css_groups
+  */
+ struct list_head cont_link_list;
+ /*
+  * List running through cg_container_links pointing at a
+  * single css_group object, anchored on css_group->cg_links
+  */
+ struct list_head cg_link_list;
+ struct css_group *cg;
+};
+
+/* The default css_group - used by init and its children prior to any
+ * hierarchies being mounted. It contains a pointer to the root state
+ * for each subsystem. Also used to anchor the list of css_groups. Not
+ * reference-counted, to improve performance when child containers
+ * haven't been created.
+ */
+
+static struct css_group init_css_group;
+static struct cg_container_link init_css_group_link;
+
+/* css_group_lock protects the list of css_group objects, and the
+ * chain of tasks off each css_group. Nests inside task->alloc_lock */
+static DEFINE_RWLOCK(css_group_lock);
+static int css_group_count;
+
+
+/* When we create or destroy a css_group, the operation simply
+ * takes/releases a reference count on all the containers referenced
+ * by subsystems in this css_group. This can end up multiple-counting
+ * some containers, but that's OK - the ref-count is just a
+ * busy/not-busy indicator; ensuring that we only count each container
+ * once would require taking a global lock to ensure that no
  * subsystems moved between hierarchies while we were doing so.
  *
  * Possible TODO: decide at boot time based on the number of
@@ -146,20 +178,218 @@ list_for_each_entry(_root, &roots, root_
  * take a global lock and only add one ref count to each hierarchy.
  */

-static void get_css_group(struct css_group *cg)
+/*
+ * unlink a css_group from the list and free it
+ */
+static void release_css_group(struct kref *k)
+{
+ struct css_group *cg =
```

```
+  container_of(k, struct css_group, ref);
+ int i;
+ write_lock(&css_group_lock);
+ list_del(&cg->list);
+ css_group_count--;
+ while (!list_empty(&cg->cg_links)) {
+  struct cg_container_link *link;
+  link = list_entry(cg->cg_links.next,
+       struct cg_container_link, cg_link_list);
+  list_del(&link->cg_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+ write_unlock(&css_group_lock);
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+  atomic_dec(&cg->subsys[i]->container->count);
+ }
+ kfree(cg);
+}
+
+/*
+ * refcounted get/put for css_group objects
+ */
+static inline void get_css_group(struct css_group *cg)
+{
+ kref_get(&cg->ref);
+}
+
+static inline void put_css_group(struct css_group *cg)
+{
+ kref_put(&cg->ref, release_css_group);
+}
+
+/*
+ * find_existing_css_group() is a helper for
+ * find_css_group(), and checks to see whether an existing
+ * css_group is suitable. This currently walks a linked-list for
+ * simplicity; a later patch will use a hash table for better
+ * performance
+ *
+ * oldcg: the container group that we're using before the container
+ * transition
+ *
+ * cont: the container that we're moving into
+ *
+ * template: location in which to build the desired set of subsystem
+ * state objects for the new container group
+ */
```

```
+
+static struct css_group *find_existing_css_group(
+ struct css_group *oldcg,
+ struct container *cont,
+ struct container_subsys_state *template[])
 {
  int i;
+ struct containerfs_root *root = cont->root;
+ struct list_head *l = &init_css_group.list;
+
+ /* Built the set of subsystem state objects that we want to
+  * see in the new css_group */
  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
-  atomic_inc(&cg->subsys[i]->container->count);
+  if (root->subsys_bits & (1ull << i)) {
+   /* Subsystem is in this hierarchy. So we want
+    * the subsystem state from the new
+    * container */
+   template[i] = cont->subsys[i];
+  } else {
+   /* Subsystem is not in this hierarchy, so we
+    * don't want to change the subsystem state */
+   template[i] = oldcg->subsys[i];
+  }
  }
+
+ /* Look through existing container groups to find one to reuse */
+ do {
+  struct css_group *cg =
+   list_entry(l, struct css_group, list);
+
+  if (!memcmp(template, cg->subsys, sizeof(cg->subsys))) {
+   /* All subsystems matched */
+   return cg;
+  }
+  /* Try the next container group */
+  l = l->next;
+ } while (l != &init_css_group.list);
+
+ /* No existing container group matched */
+ return NULL;
 }

-static void put_css_group(struct css_group *cg)
+/*
+ * allocate_cg_links() allocates "count" cg_container_link structures
+ * and chains them on tmp through their cont_link_list fields. Returns 0 on
+ * success or a negative error
```

```
+ */
+
+static int allocate_cg_links(int count, struct list_head *tmp)
 {
+ struct cg_container_link *link;
  int i;
+ INIT_LIST_HEAD(tmp);
+ for (i = 0; i < count; i++) {
+  link = kmalloc(sizeof(*link), GFP_KERNEL);
+  if (!link) {
+   while (!list_empty(tmp)) {
+    link = list_entry(tmp->next,
+        struct cg_container_link,
+        cont_link_list);
+    list_del(&link->cont_link_list);
+    kfree(link);
+   }
+   return -ENOMEM;
+  }
+  list_add(&link->cont_link_list, tmp);
+ }
+ return 0;
+}
+
+/*
+ * find_css_group() takes an existing container group and a
+ * container object, and returns a css_group object that's
+ * equivalent to the old group, but with the given container
+ * substituted into the appropriate hierarchy. Must be called with
+ * container_mutex held
+ */
+
+static struct css_group *find_css_group(
+ struct css_group *oldcg, struct container *cont)
+{
+ struct css_group *res;
+ struct container_subsys_state *template[CONTAINER_SUBSYS_COUNT];
+ int i;
+
+ struct list_head tmp_cg_links;
+ struct cg_container_link *link;
+
+ /* First see if we already have a container group that matches
+  * the desired set */
+ write_lock(&css_group_lock);
+ res = find_existing_css_group(oldcg, cont, template);
+ if (res)
+  get_css_group(res);
```

```
+ write_unlock(&css_group_lock);
+
+ if (res)
+  return res;
+
+ res = kmalloc(sizeof(*res), GFP_KERNEL);
+ if (!res)
+  return NULL;
+
+ /* Allocate all the cg_container_link objects that we'll need */
+ if (allocate_cg_links(root_count, &tmp_cg_links) < 0) {
+  kfree(res);
+  return NULL;
+ }
+
+ kref_init(&res->ref);
+ INIT_LIST_HEAD(&res->cg_links);
+ INIT_LIST_HEAD(&res->tasks);
+
+ /* Copy the set of subsystem state objects generated in
+  * find_existing_css_group() */
+ memcpy(res->subsys, template, sizeof(res->subsys));
+
+ write_lock(&css_group_lock);
+ /* Add reference counts and links from the new css_group. */
  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
-  atomic_dec(&cg->subsys[i]->container->count);
+  struct container *cont = res->subsys[i]->container;
+  struct container_subsys *ss = subsys[i];
+  atomic_inc(&cont->count);
+  /*
+   * We want to add a link once per container, so we
+   * only do it for the first subsystem in each
+   * hierarchy
+   */
+  if (ss->root->subsys_list.next == &ss->sibling) {
+   BUG_ON(list_empty(&tmp_cg_links));
+   link = list_entry(tmp_cg_links.next,
+       struct cg_container_link,
+       cont_link_list);
+   list_del(&link->cont_link_list);
+   list_add(&link->cont_link_list, &cont->css_groups);
+   link->cg = res;
+   list_add(&link->cg_link_list, &res->cg_links);
+  }
  }
+ if (list_empty(&rootnode.subsys_list)) {
+  link = list_entry(tmp_cg_links.next,
```

```
+       struct cg_container_link,
+       cont_link_list);
+  list_del(&link->cont_link_list);
+  list_add(&link->cont_link_list, &dummytop->css_groups);
+  link->cg = res;
+  list_add(&link->cg_link_list, &res->cg_links);
+ }
+
+ BUG_ON(!list_empty(&tmp_cg_links));
+
+ /* Link this container group into the list */
+ list_add(&res->list, &init_css_group.list);
+ css_group_count++;
+ INIT_LIST_HEAD(&res->tasks);
+ write_unlock(&css_group_lock);
+
+ return res;
 }

 /*
@@ -438,7 +665,19 @@ static void container_put_super(struct s
  ret = rebind_subsystems(root, 0);
  BUG_ON(ret);

+ write_lock(&css_group_lock);
+ while (!list_empty(&cont->css_groups)) {
+  struct cg_container_link *link;
+  link = list_entry(cont->css_groups.next,
+       struct cg_container_link, cont_link_list);
+  list_del(&link->cg_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+ write_unlock(&css_group_lock);
+
  list_del(&root->root_list);
+ root_count--;
  kfree(root);
  mutex_unlock(&container_mutex);
 }
@@ -568,7 +807,9 @@ static void init_container_root(struct c
  cont->top_container = cont;
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
+ INIT_LIST_HEAD(&cont->css_groups);
  list_add(&root->root_list, &roots);
+ root_count++;
 }
```

```
 static int container_get_sb(struct file_system_type *fs_type,
@@ -604,12 +845,55 @@ static int container_get_sb(struct file_

  if (!use_existing) {
   /* We need a new root */
+  struct list_head tmp_cg_links, *l;
   root = kzalloc(sizeof(*root), GFP_KERNEL);
   if (!root) {
    ret = -ENOMEM;
    goto out_unlock;
   }
+  /* We're accessing css_group_count without locking
+   * here, but that's OK - it can only be increased by
+   * someone holding container_lock, and that's us. The
+   * worst that can happen is that we have some link
+   * structures left over */
+  ret = allocate_cg_links(css_group_count, &tmp_cg_links);
+  if (ret < 0) {
+   kfree(root);
+   goto out_unlock;
+  }
   init_container_root(root);
+
+  /* Link the top container in this hierarchy into all
+   * the css_group objects */
+  write_lock(&css_group_lock);
+  l = &init_css_group.list;
+  do {
+   struct css_group *cg;
+   struct cg_container_link *link;
+   cg = list_entry(l, struct css_group, list);
+   BUG_ON(list_empty(&tmp_cg_links));
+   link = list_entry(tmp_cg_links.next,
+       struct cg_container_link,
+       cont_link_list);
+   list_del(&link->cont_link_list);
+   link->cg = cg;
+   list_add(&link->cont_link_list,
+     &root->top_container.css_groups);
+   list_add(&link->cg_link_list, &cg->cg_links);
+   l = l->next;
+  } while (l != &init_css_group.list);
+  write_unlock(&css_group_lock);
+
+  while (!list_empty(&tmp_cg_links)) {
+   /* Probably shouldn't happen */
+   struct cg_container_link *link;
```

```
+   printk(KERN_INFO "Freeing unused cg_container_link\n");
+   link = list_entry(tmp_cg_links.next,
+       struct cg_container_link,
+       cont_link_list);
+   list_del(&link->cont_link_list);
+   kfree(link);
+  }
  }

  if (!root->sb) {
@@ -722,9 +1006,9 @@ static int attach_task(struct container
  int retval = 0;
  struct container_subsys *ss;
  struct container *oldcont;
- struct css_group *cg = &tsk->containers;
+ struct css_group *cg = tsk->containers;
+ struct css_group *newcg;
  struct containerfs_root *root = cont->root;
- int i;

  int subsys_id;
  get_first_subsys(cont, NULL, &subsys_id);
@@ -743,23 +1027,25 @@ static int attach_task(struct container
   }
  }

+ /* Locate or allocate a new css_group for this task,
+  * based on its final set of containers */
+ newcg = find_css_group(cg, cont);
+ if (!newcg) {
+  return -ENOMEM;
+ }
+
  task_lock(tsk);
  if (tsk->flags & PF_EXITING) {
   task_unlock(tsk);
+  put_css_group(newcg);
   return -ESRCH;
  }
- /* Update the css_group pointers for the subsystems in this
-  * hierarchy */
- for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
-  if (root->subsys_bits & (1ull << i)) {
-   /* Subsystem is in this hierarchy. So we want
-    * the subsystem state from the new
-    * container. Transfer the refcount from the
-    * old to the new */
-   atomic_inc(&cont->count);
```

```
-   atomic_dec(&cg->subsys[i]->container->count);
-   rcu_assign_pointer(cg->subsys[i], cont->subsys[i]);
-  }
+ rcu_assign_pointer(tsk->containers, newcg);
+ if (!list_empty(&tsk->cg_list)) {
+  write_lock(&css_group_lock);
+  list_del(&tsk->cg_list);
+  list_add(&tsk->cg_list, &newcg->tasks);
+  write_unlock(&css_group_lock);
  }
  task_unlock(tsk);

@@ -770,6 +1056,7 @@ static int attach_task(struct container
  }

  synchronize_rcu();
+ put_css_group(cg);
  return 0;
 }

@@ -1057,27 +1344,79 @@ int container_add_files(struct container
  return 0;
 }

-/* Count the number of tasks in a container. Could be made more
- * time-efficient but less space-efficient with more linked lists
- * running through each container and the css_group structures that
- * referenced it. Must be called with tasklist_lock held for read or
- * write or in an rcu critical section. */
+/* Count the number of tasks in a container. */

-int __container_task_count(const struct container *cont)
+int container_task_count(const struct container *cont)
 {
  int count = 0;
- struct task_struct *g, *p;
- struct container_subsys_state *css;
- int subsys_id;
- get_first_subsys(cont, &css, &subsys_id);
+ struct list_head *l;

- do_each_thread(g, p) {
-  if (task_subsys_state(p, subsys_id) == css)
-   count ++;
- } while_each_thread(g, p);
+ read_lock(&css_group_lock);
+ l = cont->css_groups.next;
+ while (l != &cont->css_groups) {
```

```
+  struct cg_container_link *link =
+   list_entry(l, struct cg_container_link, cont_link_list);
+  count += atomic_read(&link->cg->ref.refcount);
+  l = l->next;
+ }
+ read_unlock(&css_group_lock);
  return count;
 }

+/* Advance a list_head iterator pointing at a cg_container_link's */
+static inline void container_advance_iter(struct container *cont,
+       struct container_iter *it)
+{
+ struct list_head *l = it->cg_link;
+ struct cg_container_link *link;
+ struct css_group *cg;
+ /* Advance to the next non-empty css_group */
+ do {
+  l = l->next;
+  if (l == &cont->css_groups) {
+   it->cg_link = NULL;
+   return;
+  }
+  link = list_entry(l, struct cg_container_link, cont_link_list);
+  cg = link->cg;
+ } while (list_empty(&cg->tasks));
+ it->cg_link = l;
+ it->task = cg->tasks.next;
+}
+
+void container_iter_start(struct container *cont, struct container_iter *it)
+{
+ read_lock(&css_group_lock);
+ it->cg_link = &cont->css_groups;
+ container_advance_iter(cont, it);
+}
+
+struct task_struct *container_iter_next(struct container *cont,
+     struct container_iter *it)
+{
+ struct task_struct *res;
+ struct list_head *l = it->task;
+ /* If the iterator cg is NULL, we have no tasks */
+ if (!it->cg_link)
+  return NULL;
+ res = list_entry(l, struct task_struct, cg_list);
+ /* Advance iterator to find next entry */
+ l = l->next;
```

```
+ if (l == &res->containers->tasks) {
+  /* We reached the end of this task list - move on to
+   * the next cg_container_link */
+  container_advance_iter(cont, it);
+ } else {
+  it->task = l;
+ }
+ return res;
+}
+
+void container_iter_end(struct container *cont, struct container_iter *it)
+{
+ read_unlock(&css_group_lock);
+}
+
 /*
  * Stuff for reading the 'tasks' file.
  *
@@ -1110,22 +1449,15 @@ struct ctr_struct {
 static int pid_array_load(pid_t *pidarray, int npids, struct container *cont)
 {
  int n = 0;
- struct task_struct *g, *p;
- struct container_subsys_state *css;
- int subsys_id;
- get_first_subsys(cont, &css, &subsys_id);
- rcu_read_lock();
-
- do_each_thread(g, p) {
-  if (task_subsys_state(p, subsys_id) == css) {
-   pidarray[n++] = pid_nr(task_pid(p));
-   if (unlikely(n == npids))
-    goto array_full;
-  }
- } while_each_thread(g, p);
-
-array_full:
- rcu_read_unlock();
+ struct container_iter it;
+ struct task_struct *tsk;
+ container_iter_start(cont, &it);
+ while ((tsk = container_iter_next(cont, &it))) {
+  pidarray[n++] = pid_nr(task_pid(tsk));
+  if (unlikely(n == npids))
+   break;
+ }
+ container_iter_end(cont, &it);
  return n;
```

```
 }

@@ -1299,6 +1631,7 @@ static long container_create(struct cont
  cont->flags = 0;
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
+ INIT_LIST_HEAD(&cont->css_groups);

  cont->parent = parent;
  cont->root = parent->root;
@@ -1428,8 +1761,8 @@ static int container_rmdir(struct inode
 static void container_init_subsys(struct container_subsys *ss)
 {
  int retval;
- struct task_struct *g, *p;
  struct container_subsys_state *css;
+ struct list_head *l;
  printk(KERN_ERR "Initializing container subsys %s\n", ss->name);

  /* Create the top container state for this subsystem */
@@ -1440,26 +1773,32 @@ static void container_init_subsys(struct
  init_container_css(ss, dummytop);
  css = dummytop->subsys[ss->subsys_id];

- /* Update all tasks to contain a subsys pointer to this state
-  * - since the subsystem is newly registered, all tasks are in
-  * the subsystem's top container. */
+ /* Update all container groups to contain a subsys
+  * pointer to this state - since the subsystem is
+  * newly registered, all tasks and hence all container
+  * groups are in the subsystem's top container. */
+ write_lock(&css_group_lock);
+ l = &init_css_group.list;
+ do {
+  struct css_group *cg =
+   list_entry(l, struct css_group, list);
+  cg->subsys[ss->subsys_id] = dummytop->subsys[ss->subsys_id];
+  l = l->next;
+ } while (l != &init_css_group.list);
+ write_unlock(&css_group_lock);

   /* If this subsystem requested that it be notified with fork
    * events, we should send it one now for every process in the
    * system */
+  if (ss->fork) {
+   struct task_struct *g, *p;

- read_lock(&tasklist_lock);
```

```
- init_task.containers.subsys[ss->subsys_id] = css;
- if (ss->fork)
-  ss->fork(ss, &init_task);
-
- do_each_thread(g, p) {
-  printk(KERN_INFO "Setting task %p css to %p (%d)\n", css, p, p->pid);
-  p->containers.subsys[ss->subsys_id] = css;
-  if (ss->fork)
-   ss->fork(ss, p);
- } while_each_thread(g, p);
- read_unlock(&tasklist_lock);
+   read_lock(&tasklist_lock);
+   do_each_thread(g, p) {
+    ss->fork(ss, p);
+   } while_each_thread(g, p);
+   read_unlock(&tasklist_lock);
+  }

  need_forkexit_callback |= ss->fork || ss->exit;

@@ -1475,7 +1814,20 @@ static void container_init_subsys(struct
 int __init container_init_early(void)
 {
  int i;
+ kref_init(&init_css_group.ref);
+ kref_get(&init_css_group.ref);
+ INIT_LIST_HEAD(&init_css_group.list);
+ INIT_LIST_HEAD(&init_css_group.cg_links);
+ INIT_LIST_HEAD(&init_css_group.tasks);
+ css_group_count = 1;
  init_container_root(&rootnode);
+ init_task.containers = &init_css_group;
+
+ init_css_group_link.cg = &init_css_group;
+ list_add(&init_css_group_link.cont_link_list,
+   &rootnode.top_container.css_groups);
+ list_add(&init_css_group_link.cg_link_list,
+   &init_css_group.cg_links);

  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
   struct container_subsys *ss = subsys[i];
@@ -1632,6 +1984,7 @@ static int proc_containerstats_show(stru
   seq_printf(m, "%d: name=%s hierarchy=%p\n",
       i, ss->name, ss->root);
  }
+ seq_printf(m, "Container groups: %d\n", css_group_count);
  mutex_unlock(&container_mutex);
  return 0;
```

```
 }
@@ -1657,8 +2010,8 @@ struct file_operations proc_containersta
  * fork.c by dup_task_struct().  However, we ignore that copy, since
  * it was not made under the protection of RCU or container_mutex, so
  * might no longer be a valid container pointer.  attach_task() might
- * have already changed current->container, allowing the previously
- * referenced container to be removed and freed.
+ * have already changed current->containers, allowing the previously
+ * referenced container group to be removed and freed.
  *
  * At the point that container_fork() is called, 'current' is the parent
  * task, and the passed argument 'child' points to the child task.
@@ -1666,10 +2019,11 @@ struct file_operations proc_containersta

 void container_fork(struct task_struct *child)
 {
- rcu_read_lock();
- child->containers = rcu_dereference(current->containers);
- get_css_group(&child->containers);
- rcu_read_unlock();
+ write_lock(&css_group_lock);
+ child->containers = current->containers;
+ get_css_group(child->containers);
+ list_add(&child->cg_list, &child->containers->tasks);
+ write_unlock(&css_group_lock);
 }

 /**
@@ -1730,6 +2084,7 @@ void container_fork_callbacks(struct tas
 void container_exit(struct task_struct *tsk, int run_callbacks)
 {
  int i;
+ struct css_group *cg = NULL;
  if (run_callbacks && need_forkexit_callback) {
   for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
    struct container_subsys *ss = subsys[i];
@@ -1738,11 +2093,18 @@ void container_exit(struct task_struct *
    }
   }
  }
+
  /* Reassign the task to the init_css_group. */
  task_lock(tsk);
- put_css_group(&tsk->containers);
- tsk->containers = init_task.containers;
+ write_lock(&css_group_lock);
+ list_del(&tsk->cg_list);
+ write_unlock(&css_group_lock);
```

```
+
+ cg = tsk->containers;
+ tsk->containers = &init_css_group;
  task_unlock(tsk);
+ if (cg)
+  put_css_group(cg);
 }

 static atomic_t namecnt;
@@ -1781,11 +2143,13 @@ int container_clone(struct task_struct *
   mutex_unlock(&container_mutex);
   return 0;
  }
- cg = &tsk->containers;
+ cg = tsk->containers;
  parent = task_container(tsk, subsys->subsys_id);
  /* Pin the hierarchy */
  atomic_inc(&parent->root->sb->s_active);

+ /* Keep the container alive */
+ get_css_group(cg);
  mutex_unlock(&container_mutex);

  /* Now do the VFS work to create a container */
@@ -1830,6 +2194,7 @@ int container_clone(struct task_struct *
      (parent != task_container(tsk, subsys->subsys_id))) {
   /* Aargh, we raced ... */
   mutex_unlock(&inode->i_mutex);
+  put_css_group(cg);

   deactivate_super(parent->root->sb);
   /* The container is still accessible in the VFS, but
@@ -1847,6 +2212,7 @@ int container_clone(struct task_struct *

  out_release:
  mutex_unlock(&inode->i_mutex);
+ put_css_group(cg);
  deactivate_super(parent->root->sb);
  return ret;
 }
Index: container-2.6.22-rc2-mm1/kernel/cpuset.c
 ===================================================================
--- container-2.6.22-rc2-mm1.orig/kernel/cpuset.c
+++ container-2.6.22-rc2-mm1/kernel/cpuset.c
@@ -593,12 +593,13 @@ static int update_nodemask(struct cpuset
 {
  struct cpuset trialcs;
  nodemask_t oldmem;
```

```
- struct task_struct *g, *p;
+ struct task_struct *p;
  struct mm_struct **mmarray;
  int i, n, ntasks;
  int migrate;
  int fudge;
  int retval;
+ struct container_iter it;

  /* top_cpuset.mems_allowed tracks node_online_map; it's read-only */
  if (cs == &top_cpuset)
@@ -659,7 +660,7 @@ static int update_nodemask(struct cpuset
   if (!mmarray)
    goto done;
   read_lock(&tasklist_lock);  /* block fork */
-  if (__container_task_count(cs->css.container) <= ntasks)
+  if (container_task_count(cs->css.container) <= ntasks)
    break;    /* got enough */
   read_unlock(&tasklist_lock);  /* try again */
   kfree(mmarray);
@@ -668,21 +669,21 @@ static int update_nodemask(struct cpuset
  n = 0;

  /* Load up mmarray[] with mm reference for each task in cpuset. */
- do_each_thread(g, p) {
+ container_iter_start(cs->css.container, &it);
+ while ((p = container_iter_next(cs->css.container, &it))) {
   struct mm_struct *mm;

   if (n >= ntasks) {
    printk(KERN_WARNING
     "Cpuset mempolicy rebind incomplete.\n");
-   continue;
+   break;
   }
-  if (task_cs(p) != cs)
-   continue;
   mm = get_task_mm(p);
   if (!mm)
    continue;
   mmarray[n++] = mm;
- } while_each_thread(g, p);
+ }
+ container_iter_end(cs->css.container, &it);
  read_unlock(&tasklist_lock);

  /*
Index: container-2.6.22-rc2-mm1/Documentation/containers.txt
```

```
 ========================================================== =======
--- container-2.6.22-rc2-mm1.orig/Documentation/containers.txt
+++ container-2.6.22-rc2-mm1/Documentation/containers.txt
@@ -176,7 +176,9 @@ Containers extends the kernel as follows
    subsystem state is something that's expected to happen frequently
    and in performance-critical code, whereas operations that require a
    task's actual container assignments (in particular, moving between
-   containers) are less common.
+   containers) are less common. A linked list runs through the cg_list
+   field of each task_struct using the css_group, anchored at
+   css_group->tasks.

  - A container hierarchy filesystem can be mounted  for browsing and
    manipulation from user space.
@@ -252,6 +254,16 @@ linear search to locate an appropriate e
 very efficient. A future version will use a hash table for better
 performance.

+To allow access from a container to the css_groups (and hence tasks)
+that comprise it, a set of cg_container_link objects form a lattice;
+each cg_container_link is linked into a list of cg_container_links for
+a single container on its cont_link_list field, and a list of
+cg_container_links for a single css_group on its cg_link_list.
+
+Thus the set of tasks in a container can be listed by iterating over
+each css_group that references the container, and sub-iterating over
+each css_group's task set.
+
 The use of a Linux virtual file system (vfs) to represent the
 container hierarchy provides for a familiar permission and name space
 for containers, with a minimum of additional kernel code.

--
```