Subject: [PATCH 07/33] task containersv11 shared container subsystem group arrays
Posted by Paul Menage on Mon, 17 Sep 2007 21:03:14 GMT
View Forum Message <> Reply to Message

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

Assuming that many tasks share the same cgroup 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-cgroups kernel, no matter how many
subsystems are registered).

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

```
 Documentation/cgroups.txt |   14
 include/linux/cgroup.h    |   89 ++++
 include/linux/sched.h     |   33 -
 kernel/cgroup.c           |  606 ++++++++++++++++++++++++++++++-----
 kernel/fork.c             |    1
 5 files changed, 620 insertions(+), 123 deletions(-)

diff -puN Documentation/cgroups.txt~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
Documentation/cgroups.txt
--- a/Documentation/cgroups.txt~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
+++ a/Documentation/cgroups.txt
@@ -176,7 +176,9 @@ Control Groups 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 cgroup assignments (in particular, moving between
-  cgroups) are less common.
+  cgroups) are less common. A linked list runs through the cg_list
+  field of each task_struct using the css_set, anchored at
+  css_set->tasks.

 - A cgroup 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 cgroup to the css_sets (and hence tasks)
+that comprise it, a set of cg_cgroup_link objects form a lattice;
+each cg_cgroup_link is linked into a list of cg_cgroup_links for
+a single cgroup on its cont_link_list field, and a list of
```

```
+cg_cgroup_links for a single css_set on its cg_link_list.
+
+Thus the set of tasks in a cgroup can be listed by iterating over
+each css_set that references the cgroup, and sub-iterating over
+each css_set's task set.
+
 The use of a Linux virtual file system (vfs) to represent the
 cgroup hierarchy provides for a familiar permission and name space
 for cgroups, with a minimum of additional kernel code.
diff -puN include/linux/cgroup.h~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
include/linux/cgroup.h
--- a/include/linux/cgroup.h~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
+++ a/include/linux/cgroup.h
@@ -27,10 +27,19 @@ extern void cgroup_lock(void);
 extern void cgroup_unlock(void);
 extern void cgroup_fork(struct task_struct *p);
 extern void cgroup_fork_callbacks(struct task_struct *p);
+extern void cgroup_post_fork(struct task_struct *p);
 extern void cgroup_exit(struct task_struct *p, int run_callbacks);

 extern struct file_operations proc_cgroup_operations;

+/* Define the enumeration of all cgroup subsystems */
+#define SUBSYS(_x) _x ## _subsys_id,
+enum cgroup_subsys_id {
+#include <linux/cgroup_subsys.h>
+ CGROUP_SUBSYS_COUNT
+};
+#undef SUBSYS
+
 /* Per-subsystem/per-cgroup state maintained by the system. */
 struct cgroup_subsys_state {
  /* The cgroup that this subsystem is attached to. Useful
@@ -97,6 +106,52 @@ struct cgroup {

  struct cgroupfs_root *root;
  struct cgroup *top_cgroup;
+
+ /*
+  * List of cg_cgroup_links pointing at css_sets with
+  * tasks in this cgroup. Protected by css_set_lock
+  */
+ struct list_head css_sets;
+};
+
+/* A css_set is a structure holding pointers to a set of
+ * cgroup_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
+ * cgroup set for a task.
+ */
+
+struct css_set {
+
+ /* Reference count */
+ struct kref ref;
+
+ /*
+  * List running through all cgroup groups. Protected by
+  * css_set_lock
+  */
+ struct list_head list;
+
+ /*
+  * List running through all tasks using this cgroup
+  * group. Protected by css_set_lock
+  */
+ struct list_head tasks;
+
+ /*
+  * List of cg_cgroup_link objects on link chains from
+  * cgroups referenced from this css_set. Protected by
+  * css_set_lock
+  */
+ struct list_head cg_links;
+
+ /*
+  * Set of subsystem states, one for each subsystem. This array
+  * is immutable after creation apart from the init_css_set
+  * during subsystem registration (at boot time).
+  */
+ struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
+
 };

 /* struct cftype:
@@ -149,15 +204,7 @@ int cgroup_is_removed(const struct co

 int cgroup_path(const struct cgroup *cont, char *buf, int buflen);

-int __cgroup_task_count(const struct cgroup *cont);
-static inline int cgroup_task_count(const struct cgroup *cont)
-{
- int task_count;
- rcu_read_lock();
- task_count = __cgroup_task_count(cont);
```

```
- rcu_read_unlock();
- return task_count;
-}
+int cgroup_task_count(const struct cgroup *cont);

 /* Return true if the cgroup is a descendant of the current cgroup */
 int cgroup_is_descendant(const struct cgroup *cont);
@@ -205,7 +252,7 @@ static inline struct cgroup_subsys_st
 static inline struct cgroup_subsys_state *task_subsys_state(
  struct task_struct *task, int subsys_id)
 {
- return rcu_dereference(task->cgroups.subsys[subsys_id]);
+ return rcu_dereference(task->cgroups->subsys[subsys_id]);
 }

 static inline struct cgroup* task_cgroup(struct task_struct *task,
@@ -218,6 +265,27 @@ int cgroup_path(const struct containe

 int cgroup_clone(struct task_struct *tsk, struct cgroup_subsys *ss);

+/* A cgroup_iter should be treated as an opaque object */
+struct cgroup_iter {
+ struct list_head *cg_link;
+ struct list_head *task;
+};
+
+/* To iterate across the tasks in a cgroup:
+ *
+ * 1) call cgroup_iter_start to intialize an iterator
+ *
+ * 2) call cgroup_iter_next() to retrieve member tasks until it
+ *    returns NULL or until you want to end the iteration
+ *
+ * 3) call cgroup_iter_end() to destroy the iterator.
+ */
+void cgroup_iter_start(struct cgroup *cont, struct cgroup_iter *it);
+struct task_struct *cgroup_iter_next(struct cgroup *cont,
+     struct cgroup_iter *it);
+void cgroup_iter_end(struct cgroup *cont, struct cgroup_iter *it);
+
+
 #else /* !CONFIG_CGROUPS */

 static inline int cgroup_init_early(void) { return 0; }
@@ -225,6 +293,7 @@ static inline int cgroup_init(void) {
 static inline void cgroup_init_smp(void) {}
 static inline void cgroup_fork(struct task_struct *p) {}
 static inline void cgroup_fork_callbacks(struct task_struct *p) {}
```

```
+static inline void cgroup_post_fork(struct task_struct *p) {}
 static inline void cgroup_exit(struct task_struct *p, int callbacks) {}

 static inline void cgroup_lock(void) {}
diff -puN include/linux/sched.h~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
include/linux/sched.h
--- a/include/linux/sched.h~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
+++ a/include/linux/sched.h
@@ -861,34 +861,6 @@ struct sched_entity {
 #endif
 };

-#ifdef CONFIG_CGROUPS
-
-#define SUBSYS(_x) _x ## _subsys_id,
-enum cgroup_subsys_id {
-#include <linux/cgroup_subsys.h>
- CGROUP_SUBSYS_COUNT
-};
-#undef SUBSYS
-
-/* A css_set is a structure holding pointers to a set of
- * cgroup_subsys_state objects.
- */
-
-struct css_set {
-
- /* 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 cgroup, but result
-  * in increased space usage if tasks are in wildly different
-  * groupings across different hierarchies. This array is
-  * immutable after creation */
- struct cgroup_subsys_state *subsys[CGROUP_SUBSYS_COUNT];
-
-};
-
-#endif /* CONFIG_CGROUPS */
-
 struct task_struct {
  volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
@@ -1125,7 +1097,10 @@ struct task_struct {
  int cpuset_mem_spread_rotor;
 #endif
 #ifdef CONFIG_CGROUPS
- struct css_set cgroups;
```

```
+ /* Control Group info protected by css_set_lock */
+ struct css_set *cgroups;
+ /* cg_list protected by css_set_lock and tsk->alloc_lock */
+ struct list_head cg_list;
 #endif
 #ifdef CONFIG_FUTEX
  struct robust_list_head __user *robust_list;
diff -puN kernel/cgroup.c~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
kernel/cgroup.c
--- a/kernel/cgroup.c~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
+++ a/kernel/cgroup.c
@@ -95,6 +95,7 @@ static struct cgroupfs_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 cgroup */
 #define dummytop (&rootnode.top_cgroup)
@@ -133,12 +134,49 @@ 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_set, so the get/put
- * operation simply takes a reference count on all the cgroups
- * referenced by subsystems in this css_set. This can end up
- * multiple-counting some cgroups, but that's OK - the ref-count is
- * just a busy/not-busy indicator; ensuring that we only count each
- * cgroup once would require taking a global lock to ensure that no
+/* Link structure for associating css_set objects with cgroups */
+struct cg_cgroup_link {
+ /*
+  * List running through cg_cgroup_links associated with a
+  * cgroup, anchored on cgroup->css_sets
+  */
+ struct list_head cont_link_list;
+ /*
+  * List running through cg_cgroup_links pointing at a
+  * single css_set object, anchored on css_set->cg_links
+  */
+ struct list_head cg_link_list;
+ struct css_set *cg;
+};
+
+/* The default css_set - 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_sets. Not
+ * reference-counted, to improve performance when child cgroups
```

```
+ * haven't been created.
+ */
+
+static struct css_set init_css_set;
+static struct cg_cgroup_link init_css_set_link;
+
+/* css_set_lock protects the list of css_set objects, and the
+ * chain of tasks off each css_set.  Nests outside task->alloc_lock
+ * due to cgroup_iter_start() */
+static DEFINE_RWLOCK(css_set_lock);
+static int css_set_count;
+
+/* We don't maintain the lists running through each css_set to its
+ * task until after the first call to cgroup_iter_start(). This
+ * reduces the fork()/exit() overhead for people who have cgroups
+ * compiled into their kernel but not actually in use */
+static int use_task_css_set_links;
+
+/* When we create or destroy a css_set, the operation simply
+ * takes/releases a reference count on all the cgroups referenced
+ * by subsystems in this css_set. This can end up multiple-counting
+ * some cgroups, but that's OK - the ref-count is just a
+ * busy/not-busy indicator; ensuring that we only count each cgroup
+ * 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
@@ -146,18 +184,230 @@ list_for_each_entry(_root, &roots, root_
  * it's better for performance to ref-count every subsystem, or to
  * take a global lock and only add one ref count to each hierarchy.
  */
-static void get_css_set(struct css_set *cg)
+
+/*
+ * unlink a css_set from the list and free it
+ */
+static void release_css_set(struct kref *k)
 {
+ struct css_set *cg = container_of(k, struct css_set, ref);
  int i;
+
+ write_lock(&css_set_lock);
+ list_del(&cg->list);
+ css_set_count--;
+ while (!list_empty(&cg->cg_links)) {
+  struct cg_cgroup_link *link;
+  link = list_entry(cg->cg_links.next,
+      struct cg_cgroup_link, cg_link_list);
```

```
+  list_del(&link->cg_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+ write_unlock(&css_set_lock);
  for (i = 0; i < CGROUP_SUBSYS_COUNT; i++)
-  atomic_inc(&cg->subsys[i]->cgroup->count);
+  atomic_dec(&cg->subsys[i]->cgroup->count);
+ kfree(cg);
+}
+
+/*
+ * refcounted get/put for css_set objects
+ */
+static inline void get_css_set(struct css_set *cg)
+{
+ kref_get(&cg->ref);
 }

-static void put_css_set(struct css_set *cg)
+static inline void put_css_set(struct css_set *cg)
+{
+ kref_put(&cg->ref, release_css_set);
+}
+
+/*
+ * find_existing_css_set() is a helper for
+ * find_css_set(), and checks to see whether an existing
+ * css_set is suitable. This currently walks a linked-list for
+ * simplicity; a later patch will use a hash table for better
+ * performance
+ *
+ * oldcg: the cgroup group that we're using before the cgroup
+ * transition
+ *
+ * cont: the cgroup that we're moving into
+ *
+ * template: location in which to build the desired set of subsystem
+ * state objects for the new cgroup group
+ */
+
+static struct css_set *find_existing_css_set(
+ struct css_set *oldcg,
+ struct cgroup *cont,
+ struct cgroup_subsys_state *template[])
 {
  int i;
- for (i = 0; i < CGROUP_SUBSYS_COUNT; i++)
```

```
-  atomic_dec(&cg->subsys[i]->cgroup->count);
+ struct cgroupfs_root *root = cont->root;
+ struct list_head *l = &init_css_set.list;
+
+ /* Built the set of subsystem state objects that we want to
+  * see in the new css_set */
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+  if (root->subsys_bits & (1ull << i)) {
+   /* Subsystem is in this hierarchy. So we want
+    * the subsystem state from the new
+    * cgroup */
+   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 cgroup groups to find one to reuse */
+ do {
+  struct css_set *cg =
+   list_entry(l, struct css_set, list);
+
+  if (!memcmp(template, cg->subsys, sizeof(cg->subsys))) {
+   /* All subsystems matched */
+   return cg;
+  }
+  /* Try the next cgroup group */
+  l = l->next;
+ } while (l != &init_css_set.list);
+
+ /* No existing cgroup group matched */
+ return NULL;
+}
+
+/*
+ * allocate_cg_links() allocates "count" cg_cgroup_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_cgroup_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_cgroup_link,
+        cont_link_list);
+    list_del(&link->cont_link_list);
+    kfree(link);
+   }
+   return -ENOMEM;
+  }
+  list_add(&link->cont_link_list, tmp);
+ }
+ return 0;
+}
+
+static void free_cg_links(struct list_head *tmp)
+{
+ while (!list_empty(tmp)) {
+  struct cg_cgroup_link *link;
+  link = list_entry(tmp->next,
+      struct cg_cgroup_link,
+      cont_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+}
+
+/*
+ * find_css_set() takes an existing cgroup group and a
+ * cgroup object, and returns a css_set object that's
+ * equivalent to the old group, but with the given cgroup
+ * substituted into the appropriate hierarchy. Must be called with
+ * cgroup_mutex held
+ */
+
+static struct css_set *find_css_set(
+ struct css_set *oldcg, struct cgroup *cont)
+{
+ struct css_set *res;
+ struct cgroup_subsys_state *template[CGROUP_SUBSYS_COUNT];
+ int i;
+
+ struct list_head tmp_cg_links;
+ struct cg_cgroup_link *link;
+
+ /* First see if we already have a cgroup group that matches
+  * the desired set */
```

```
+ write_lock(&css_set_lock);
+ res = find_existing_css_set(oldcg, cont, template);
+ if (res)
+  get_css_set(res);
+ write_unlock(&css_set_lock);
+
+ if (res)
+  return res;
+
+ res = kmalloc(sizeof(*res), GFP_KERNEL);
+ if (!res)
+  return NULL;
+
+ /* Allocate all the cg_cgroup_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_set() */
+ memcpy(res->subsys, template, sizeof(res->subsys));
+
+ write_lock(&css_set_lock);
+ /* Add reference counts and links from the new css_set. */
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+  struct cgroup *cont = res->subsys[i]->cgroup;
+  struct cgroup_subsys *ss = subsys[i];
+  atomic_inc(&cont->count);
+  /*
+   * We want to add a link once per cgroup, 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_cgroup_link,
+       cont_link_list);
+   list_del(&link->cont_link_list);
+   list_add(&link->cont_link_list, &cont->css_sets);
+   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_cgroup_link,
+      cont_link_list);
+  list_del(&link->cont_link_list);
+  list_add(&link->cont_link_list, &dummytop->css_sets);
+  link->cg = res;
+  list_add(&link->cg_link_list, &res->cg_links);
+ }
+
+ BUG_ON(!list_empty(&tmp_cg_links));
+
+ /* Link this cgroup group into the list */
+ list_add(&res->list, &init_css_set.list);
+ css_set_count++;
+ INIT_LIST_HEAD(&res->tasks);
+ write_unlock(&css_set_lock);
+
+ return res;
 }

 /*
@@ -516,6 +766,7 @@ static void init_cgroup_root(struct c
  cont->top_cgroup = cont;
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
+ INIT_LIST_HEAD(&cont->css_sets);
 }

 static int cgroup_test_super(struct super_block *sb, void *data)
@@ -585,6 +836,8 @@ static int cgroup_get_sb(struct file_
  int ret = 0;
  struct super_block *sb;
  struct cgroupfs_root *root;
+ struct list_head tmp_cg_links, *l;
+ INIT_LIST_HEAD(&tmp_cg_links);

  /* First find the desired set of subsystems */
  ret = parse_cgroupfs_options(data, &opts);
@@ -623,6 +876,19 @@ static int cgroup_get_sb(struct file_

   mutex_lock(&cgroup_mutex);

+  /*
+   * We're accessing css_set_count without locking
+   * css_set_lock here, but that's OK - it can only be
+   * increased by someone holding cgroup_lock, and
```

```
+   * that's us. The worst that can happen is that we
+   * have some link structures left over
+   */
+  ret = allocate_cg_links(css_set_count, &tmp_cg_links);
+  if (ret) {
+   mutex_unlock(&cgroup_mutex);
+   goto drop_new_super;
+  }
+
   ret = rebind_subsystems(root, root->subsys_bits);
   if (ret == -EBUSY) {
    mutex_unlock(&cgroup_mutex);
@@ -633,10 +899,34 @@ static int cgroup_get_sb(struct file_
   BUG_ON(ret);

   list_add(&root->root_list, &roots);
+  root_count++;

   sb->s_root->d_fsdata = &root->top_cgroup;
   root->top_cgroup.dentry = sb->s_root;

+  /* Link the top cgroup in this hierarchy into all
+   * the css_set objects */
+  write_lock(&css_set_lock);
+  l = &init_css_set.list;
+  do {
+   struct css_set *cg;
+   struct cg_cgroup_link *link;
+   cg = list_entry(l, struct css_set, list);
+   BUG_ON(list_empty(&tmp_cg_links));
+   link = list_entry(tmp_cg_links.next,
+       struct cg_cgroup_link,
+       cont_link_list);
+   list_del(&link->cont_link_list);
+   link->cg = cg;
+   list_add(&link->cont_link_list,
+     &root->top_cgroup.css_sets);
+   list_add(&link->cg_link_list, &cg->cg_links);
+   l = l->next;
+  } while (l != &init_css_set.list);
+  write_unlock(&css_set_lock);
+
+  free_cg_links(&tmp_cg_links);
+
   BUG_ON(!list_empty(&cont->sibling));
   BUG_ON(!list_empty(&cont->children));
   BUG_ON(root->number_of_cgroups != 1);
@@ -659,6 +949,7 @@ static int cgroup_get_sb(struct file_
```

```
 drop_new_super:
 up_write(&sb->s_umount);
 deactivate_super(sb);
+ free_cg_links(&tmp_cg_links);
 return ret;
}

@@ -680,8 +971,25 @@ static void cgroup_kill_sb(struct sup
 /* Shouldn't be able to fail ... */
 BUG_ON(ret);

- if (!list_empty(&root->root_list))
+ /*
+  * Release all the links from css_sets to this hierarchy's
+  * root cgroup
+  */
+ write_lock(&css_set_lock);
+ while (!list_empty(&cont->css_sets)) {
+  struct cg_cgroup_link *link;
+  link = list_entry(cont->css_sets.next,
+      struct cg_cgroup_link, cont_link_list);
+  list_del(&link->cg_link_list);
+  list_del(&link->cont_link_list);
+  kfree(link);
+ }
+ write_unlock(&css_set_lock);
+
+ if (!list_empty(&root->root_list)) {
   list_del(&root->root_list);
+  root_count--;
+ }
 mutex_unlock(&cgroup_mutex);

 kfree(root);
@@ -774,9 +1082,9 @@ static int attach_task(struct cgroup
 int retval = 0;
 struct cgroup_subsys *ss;
 struct cgroup *oldcont;
- struct css_set *cg = &tsk->cgroups;
+ struct css_set *cg = tsk->cgroups;
+ struct css_set *newcg;
 struct cgroupfs_root *root = cont->root;
- int i;
 int subsys_id;

 get_first_subsys(cont, NULL, &subsys_id);
@@ -795,26 +1103,32 @@ static int attach_task(struct cgroup
  }
```

```
 }

+ /*
+  * Locate or allocate a new css_set for this task,
+  * based on its final set of cgroups
+  */
+ newcg = find_css_set(cg, cont);
+ if (!newcg) {
+  return -ENOMEM;
+ }
+
  task_lock(tsk);
  if (tsk->flags & PF_EXITING) {
   task_unlock(tsk);
+  put_css_set(newcg);
   return -ESRCH;
  }
- /* Update the css_set pointers for the subsystems in this
-  * hierarchy */
- for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
-  if (root->subsys_bits & (1ull << i)) {
-   /* Subsystem is in this hierarchy. So we want
-    * the subsystem state from the new
-    * cgroup. Transfer the refcount from the
-    * old to the new */
-   atomic_inc(&cont->count);
-   atomic_dec(&cg->subsys[i]->cgroup->count);
-   rcu_assign_pointer(cg->subsys[i], cont->subsys[i]);
-  }
- }
+ rcu_assign_pointer(tsk->cgroups, newcg);
  task_unlock(tsk);

+ /* Update the css_set linked lists if we're using them */
+ write_lock(&css_set_lock);
+ if (!list_empty(&tsk->cg_list)) {
+  list_del(&tsk->cg_list);
+  list_add(&tsk->cg_list, &newcg->tasks);
+ }
+ write_unlock(&css_set_lock);
+
  for_each_subsys(root, ss) {
   if (ss->attach) {
    ss->attach(ss, cont, oldcont, tsk);
@@ -822,6 +1136,7 @@ static int attach_task(struct cgroup
 }

 synchronize_rcu();
```

```
+ put_css_set(cg);
  return 0;
 }

@@ -1123,28 +1438,102 @@ int cgroup_add_files(struct cgroup
  return 0;
 }

-/* Count the number of tasks in a cgroup. Could be made more
- * time-efficient but less space-efficient with more linked lists
- * running through each cgroup and the css_set structures that
- * referenced it. Must be called with tasklist_lock held for read or
- * write or in an rcu critical section.
- */
-int __cgroup_task_count(const struct cgroup *cont)
+/* Count the number of tasks in a cgroup. */
+
+int cgroup_task_count(const struct cgroup *cont)
 {
  int count = 0;
- struct task_struct *g, *p;
- struct cgroup_subsys_state *css;
- int subsys_id;
+ struct list_head *l;

- get_first_subsys(cont, &css, &subsys_id);
- do_each_thread(g, p) {
-  if (task_subsys_state(p, subsys_id) == css)
-   count ++;
- } while_each_thread(g, p);
+ read_lock(&css_set_lock);
+ l = cont->css_sets.next;
+ while (l != &cont->css_sets) {
+  struct cg_cgroup_link *link =
+   list_entry(l, struct cg_cgroup_link, cont_link_list);
+  count += atomic_read(&link->cg->ref.refcount);
+  l = l->next;
+ }
+ read_unlock(&css_set_lock);
  return count;
 }

 /*
+ * Advance a list_head iterator.  The iterator should be positioned at
+ * the start of a css_set
+ */
+static void cgroup_advance_iter(struct cgroup *cont,
+       struct cgroup_iter *it)
```

```
+{
+ struct list_head *l = it->cg_link;
+ struct cg_cgroup_link *link;
+ struct css_set *cg;
+
+ /* Advance to the next non-empty css_set */
+ do {
+  l = l->next;
+  if (l == &cont->css_sets) {
+   it->cg_link = NULL;
+   return;
+  }
+  link = list_entry(l, struct cg_cgroup_link, cont_link_list);
+  cg = link->cg;
+ } while (list_empty(&cg->tasks));
+ it->cg_link = l;
+ it->task = cg->tasks.next;
+}
+
+void cgroup_iter_start(struct cgroup *cont, struct cgroup_iter *it)
+{
+ /*
+  * The first time anyone tries to iterate across a cgroup,
+  * we need to enable the list linking each css_set to its
+  * tasks, and fix up all existing tasks.
+  */
+ if (!use_task_css_set_links) {
+  struct task_struct *p, *g;
+  write_lock(&css_set_lock);
+  use_task_css_set_links = 1;
+   do_each_thread(g, p) {
+   task_lock(p);
+   if (list_empty(&p->cg_list))
+    list_add(&p->cg_list, &p->cgroups->tasks);
+   task_unlock(p);
+   } while_each_thread(g, p);
+  write_unlock(&css_set_lock);
+ }
+ read_lock(&css_set_lock);
+ it->cg_link = &cont->css_sets;
+ cgroup_advance_iter(cont, it);
+}
+
+struct task_struct *cgroup_iter_next(struct cgroup *cont,
+     struct cgroup_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->cgroups->tasks) {
+  /* We reached the end of this task list - move on to
+   * the next cg_cgroup_link */
+  cgroup_advance_iter(cont, it);
+ } else {
+  it->task = l;
+ }
+ return res;
+}
+
+void cgroup_iter_end(struct cgroup *cont, struct cgroup_iter *it)
+{
+ read_unlock(&css_set_lock);
+}
+
+/*
  * Stuff for reading the 'tasks' file.
  *
  * Reading this file can return large amounts of data if a cgroup has
@@ -1173,22 +1562,15 @@ struct ctr_struct {
 static int pid_array_load(pid_t *pidarray, int npids, struct cgroup *cont)
 {
  int n = 0;
- struct task_struct *g, *p;
- struct cgroup_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 cgroup_iter it;
+ struct task_struct *tsk;
```

```
+ cgroup_iter_start(cont, &it);
+ while ((tsk = cgroup_iter_next(cont, &it))) {
+  if (unlikely(n == npids))
+   break;
+  pidarray[n++] = pid_nr(task_pid(tsk));
+ }
+ cgroup_iter_end(cont, &it);
  return n;
 }

@@ -1373,6 +1755,7 @@ static long cgroup_create(struct cont
  cont->flags = 0;
  INIT_LIST_HEAD(&cont->sibling);
  INIT_LIST_HEAD(&cont->children);
+ INIT_LIST_HEAD(&cont->css_sets);

  cont->parent = parent;
  cont->root = parent->root;
@@ -1504,8 +1887,8 @@ static int cgroup_rmdir(struct inode

 static void cgroup_init_subsys(struct cgroup_subsys *ss)
 {
- struct task_struct *g, *p;
  struct cgroup_subsys_state *css;
+ struct list_head *l;
  printk(KERN_ERR "Initializing cgroup subsys %s\n", ss->name);

  /* Create the top cgroup state for this subsystem */
@@ -1515,26 +1898,32 @@ static void cgroup_init_subsys(struct
  BUG_ON(IS_ERR(css));
  init_cgroup_css(css, ss, dummytop);

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

@@ -1548,8 +1937,22 @@ static void cgroup_init_subsys(struct
 int __init cgroup_init_early(void)
 {
  int i;
+ kref_init(&init_css_set.ref);
+ kref_get(&init_css_set.ref);
+ INIT_LIST_HEAD(&init_css_set.list);
+ INIT_LIST_HEAD(&init_css_set.cg_links);
+ INIT_LIST_HEAD(&init_css_set.tasks);
+ css_set_count = 1;
  init_cgroup_root(&rootnode);
  list_add(&rootnode.root_list, &roots);
+ root_count = 1;
+ init_task.cgroups = &init_css_set;
+
+ init_css_set_link.cg = &init_css_set;
+ list_add(&init_css_set_link.cont_link_list,
+   &rootnode.top_cgroup.css_sets);
+ list_add(&init_css_set_link.cg_link_list,
+   &init_css_set.cg_links);
```

```
 for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
  struct cgroup_subsys *ss = subsys[i];
@@ -1707,6 +2110,7 @@ static int proc_cgroupstats_show(stru
  seq_printf(m, "%d: name=%s hierarchy=%p\n",
      i, ss->name, ss->root);
 }
+ seq_printf(m, "Control Group groups: %d\n", css_set_count);
 mutex_unlock(&cgroup_mutex);
 return 0;
}
@@ -1733,18 +2137,19 @@ static struct file_operations proc_conta
 * fork.c by dup_task_struct().  However, we ignore that copy, since
 * it was not made under the protection of RCU or cgroup_mutex, so
 * might no longer be a valid cgroup pointer.  attach_task() might
- * have already changed current->cgroup, allowing the previously
- * referenced cgroup to be removed and freed.
+ * have already changed current->cgroups, allowing the previously
+ * referenced cgroup group to be removed and freed.
 *
 * At the point that cgroup_fork() is called, 'current' is the parent
 * task, and the passed argument 'child' points to the child task.
 */
void cgroup_fork(struct task_struct *child)
{
- rcu_read_lock();
- child->cgroups = rcu_dereference(current->cgroups);
- get_css_set(&child->cgroups);
- rcu_read_unlock();
+ task_lock(current);
+ child->cgroups = current->cgroups;
+ get_css_set(child->cgroups);
+ task_unlock(current);
+ INIT_LIST_HEAD(&child->cg_list);
}

/**
@@ -1765,6 +2170,21 @@ void cgroup_fork_callbacks(struct tas
}

/**
+ * cgroup_post_fork - called on a new task after adding it to the
+ * task list. Adds the task to the list running through its css_set
+ * if necessary. Has to be after the task is visible on the task list
+ * in case we race with the first call to cgroup_iter_start() - to
+ * guarantee that the new task ends up on its list. */
+void cgroup_post_fork(struct task_struct *child)
+{
```

```
+ if (use_task_css_set_links) {
+  write_lock(&css_set_lock);
+  if (list_empty(&child->cg_list))
+   list_add(&child->cg_list, &child->cgroups->tasks);
+  write_unlock(&css_set_lock);
+ }
+}
+/**
  * cgroup_exit - detach cgroup from exiting task
  * @tsk: pointer to task_struct of exiting process
  *
@@ -1802,6 +2222,7 @@ void cgroup_fork_callbacks(struct tas
 void cgroup_exit(struct task_struct *tsk, int run_callbacks)
 {
  int i;
+ struct css_set *cg;

  if (run_callbacks && need_forkexit_callback) {
   for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
@@ -1810,11 +2231,26 @@ void cgroup_exit(struct task_struct *
     ss->exit(ss, tsk);
   }
  }
+
+ /*
+  * Unlink from the css_set task list if necessary.
+  * Optimistically check cg_list before taking
+  * css_set_lock
+  */
+ if (!list_empty(&tsk->cg_list)) {
+  write_lock(&css_set_lock);
+  if (!list_empty(&tsk->cg_list))
+   list_del(&tsk->cg_list);
+  write_unlock(&css_set_lock);
+ }
+
  /* Reassign the task to the init_css_set. */
  task_lock(tsk);
- put_css_set(&tsk->cgroups);
- tsk->cgroups = init_task.cgroups;
+ cg = tsk->cgroups;
+ tsk->cgroups = &init_css_set;
  task_unlock(tsk);
+ if (cg)
+  put_css_set(cg);
 }

 /**
```

```
@@ -1848,7 +2284,7 @@ int cgroup_clone(struct task_struct *
   mutex_unlock(&cgroup_mutex);
   return 0;
  }
- cg = &tsk->cgroups;
+ cg = tsk->cgroups;
  parent = task_cgroup(tsk, subsys->subsys_id);

  snprintf(nodename, MAX_CGROUP_TYPE_NAMELEN, "node_%d", tsk->pid);
@@ -1856,6 +2292,8 @@ int cgroup_clone(struct task_struct *
  /* Pin the hierarchy */
  atomic_inc(&parent->root->sb->s_active);

+ /* Keep the cgroup alive */
+ get_css_set(cg);
  mutex_unlock(&cgroup_mutex);

  /* Now do the VFS work to create a cgroup */
@@ -1899,6 +2337,7 @@ int cgroup_clone(struct task_struct *
      (parent != task_cgroup(tsk, subsys->subsys_id))) {
   /* Aargh, we raced ... */
   mutex_unlock(&inode->i_mutex);
+  put_css_set(cg);

   deactivate_super(parent->root->sb);
   /* The cgroup is still accessible in the VFS, but
@@ -1922,6 +2361,7 @@ int cgroup_clone(struct task_struct *

  out_release:
  mutex_unlock(&inode->i_mutex);
+ put_css_set(cg);
  deactivate_super(parent->root->sb);
  return ret;
 }
diff -puN kernel/fork.c~task-cgroupsv11-shared-cgroup-subsystem-group-arrays kernel/fork.c
--- a/kernel/fork.c~task-cgroupsv11-shared-cgroup-subsystem-group-arrays
+++ a/kernel/fork.c
@@ -1289,6 +1289,7 @@ static struct task_struct *copy_process(
   put_user(p->pid, parent_tidptr);

  proc_fork_connector(p);
+ cgroup_post_fork(p);
  return p;

 bad_fork_cleanup_namespaces:
_

--
```

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers