Subject: [PATCH 03/10] Task Containers(V11): Add fork()/exit() hooks
Posted by Paul Menage on Fri, 20 Jul 2007 18:31:55 GMT
View Forum Message <> Reply to Message

This patch adds the necessary hooks to the fork() and exit() paths to ensure
that new children inherit their parent's container assignments, and that
exiting processes release reference counts on their containers.

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

```
 include/linux/container.h |    6 ++
 kernel/container.c        |  121 ++++++++++++++++++++++++++++++++++++++++++++++++
 kernel/exit.c             |    2
 kernel/fork.c             |   14 ++++-
 4 files changed, 141 insertions(+), 2 deletions(-)

Index: container-2.6.22-rc6-mm1/include/linux/container.h
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/include/linux/container.h
+++ container-2.6.22-rc6-mm1/include/linux/container.h
@@ -25,6 +25,9 @@ extern int container_init(void);
 extern void container_init_smp(void);
 extern void container_lock(void);
 extern void container_unlock(void);
+extern void container_fork(struct task_struct *p);
+extern void container_fork_callbacks(struct task_struct *p);
+extern void container_exit(struct task_struct *p, int run_callbacks);

 /* Per-subsystem/per-container state maintained by the system. */
 struct container_subsys_state {
@@ -215,6 +218,9 @@ int container_path(const struct containe
 static inline int container_init_early(void) { return 0; }
 static inline int container_init(void) { return 0; }
 static inline void container_init_smp(void) {}
+static inline void container_fork(struct task_struct *p) {}
+static inline void container_fork_callbacks(struct task_struct *p) {}
+static inline void container_exit(struct task_struct *p, int callbacks) {}

 static inline void container_lock(void) {}
 static inline void container_unlock(void) {}
Index: container-2.6.22-rc6-mm1/kernel/container.c
 ===================================================================
--- container-2.6.22-rc6-mm1.orig/kernel/container.c
+++ container-2.6.22-rc6-mm1/kernel/container.c
@@ -132,6 +132,33 @@ 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
+ * subsystems moved between hierarchies while we were doing so.
+ *
+ * Possible TODO: decide at boot time based on the number of
+ * registered subsystems and the number of CPUs or NUMA nodes whether
+ * 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_group(struct css_group *cg)
+{
+ int i;
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++)
+  atomic_inc(&cg->subsys[i]->container->count);
+}
+
+static void put_css_group(struct css_group *cg)
+{
+ int i;
+ for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++)
+  atomic_dec(&cg->subsys[i]->container->count);
+}
+
 /*
  * There is one global container mutex. We also require taking
  * task_lock() when dereferencing a task's container subsys pointers.
@@ -1554,3 +1581,97 @@ int __init container_init(void)
 out:
  return err;
 }
+
+/**
+ * container_fork - attach newly forked task to its parents container.
+ * @tsk: pointer to task_struct of forking parent process.
+ *
+ * Description: A task inherits its parent's container at fork().
+ *
+ * A pointer to the shared css_group was automatically copied in
+ * 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.
```

```
+ *
+ * At the point that container_fork() is called, 'current' is the parent
+ * task, and the passed argument 'child' points to the child task.
+ */
+void container_fork(struct task_struct *child)
+{
+ rcu_read_lock();
+ child->containers = rcu_dereference(current->containers);
+ get_css_group(&child->containers);
+ rcu_read_unlock();
+}
+
+/**
+ * container_fork_callbacks - called on a new task very soon before
+ * adding it to the tasklist. No need to take any locks since no-one
+ * can be operating on this task
+ */
+void container_fork_callbacks(struct task_struct *child)
+{
+ if (need_forkexit_callback) {
+  int i;
+  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+   struct container_subsys *ss = subsys[i];
+   if (ss->fork)
+    ss->fork(ss, child);
+  }
+ }
+}
+
+/**
+ * container_exit - detach container from exiting task
+ * @tsk: pointer to task_struct of exiting process
+ *
+ * Description: Detach container from @tsk and release it.
+ *
+ * Note that containers marked notify_on_release force every task in
+ * them to take the global container_mutex mutex when exiting.
+ * This could impact scaling on very large systems.  Be reluctant to
+ * use notify_on_release containers where very high task exit scaling
+ * is required on large systems.
+ *
+ * the_top_container_hack:
+ *
+ *    Set the exiting tasks container to the root container (top_container).
+ *
+ *    We call container_exit() while the task is still competent to
+ *    handle notify_on_release(), then leave the task attached to the
+ *    root container in each hierarchy for the remainder of its exit.
```

```
+ *
+ *    To do this properly, we would increment the reference count on
+ *    top_container, and near the very end of the kernel/exit.c do_exit()
+ *    code we would add a second container function call, to drop that
+ *    reference.  This would just create an unnecessary hot spot on
+ *    the top_container reference count, to no avail.
+ *
+ *    Normally, holding a reference to a container without bumping its
+ *    count is unsafe.   The container could go away, or someone could
+ *    attach us to a different container, decrementing the count on
+ *    the first container that we never incremented.  But in this case,
+ *    top_container isn't going away, and either task has PF_EXITING set,
+ *    which wards off any attach_task() attempts, or task is a failed
+ *    fork, never visible to attach_task.
+ *
+ */
+void container_exit(struct task_struct *tsk, int run_callbacks)
+{
+ int i;
+
+ if (run_callbacks && need_forkexit_callback) {
+  for (i = 0; i < CONTAINER_SUBSYS_COUNT; i++) {
+   struct container_subsys *ss = subsys[i];
+   if (ss->exit)
+    ss->exit(ss, tsk);
+  }
+ }
+ /* Reassign the task to the init_css_group. */
+ task_lock(tsk);
+ put_css_group(&tsk->containers);
+ tsk->containers = init_task.containers;
+ task_unlock(tsk);
+}
Index: container-2.6.22-rc6-mm1/kernel/exit.c
 ============================================================== =======
--- container-2.6.22-rc6-mm1.orig/kernel/exit.c
+++ container-2.6.22-rc6-mm1/kernel/exit.c
@@ -33,6 +33,7 @@
 #include <linux/delayacct.h>
 #include <linux/freezer.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
 #include <linux/posix-timers.h>
@@ -990,6 +991,7 @@ fastcall NORET_TYPE void do_exit(long co
  check_stack_usage();
  exit_thread();
```

```
  cpuset_exit(tsk);
+ container_exit(tsk, 1);
  exit_keys(tsk);

  if (group_dead && tsk->signal->leader)
Index: container-2.6.22-rc6-mm1/kernel/fork.c
 ================================================================ =======
--- container-2.6.22-rc6-mm1.orig/kernel/fork.c
+++ container-2.6.22-rc6-mm1/kernel/fork.c
@@ -30,6 +30,7 @@
 #include <linux/capability.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/container.h>
 #include <linux/security.h>
 #include <linux/swap.h>
 #include <linux/syscalls.h>
@@ -968,6 +969,7 @@ static struct task_struct *copy_process(
 {
  int retval;
  struct task_struct *p = NULL;
+ int container_callbacks_done = 0;

  if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
   return ERR_PTR(-EINVAL);
@@ -1068,12 +1070,13 @@ static struct task_struct *copy_process(
  p->io_wait = NULL;
  p->audit_context = NULL;
  cpuset_fork(p);
+ container_fork(p);
 #ifdef CONFIG_NUMA
   p->mempolicy = mpol_copy(p->mempolicy);
   if (IS_ERR(p->mempolicy)) {
    retval = PTR_ERR(p->mempolicy);
    p->mempolicy = NULL;
-   goto bad_fork_cleanup_cpuset;
+   goto bad_fork_cleanup_container;
   }
  mpol_fix_fork_child_flag(p);
 #endif
@@ -1183,6 +1186,12 @@ static struct task_struct *copy_process(
  /* Perform scheduler related setup. Assign this task to a CPU. */
  sched_fork(p, clone_flags);

+ /* Now that the task is set up, run container callbacks if
+  * necessary. We need to run them before the task is visible
+  * on the tasklist. */
+ container_fork_callbacks(p);
```

```
+ container_callbacks_done = 1;
+
  /* Need tasklist lock for parent etc handling! */
  write_lock_irq(&tasklist_lock);

@@ -1305,9 +1314,10 @@ bad_fork_cleanup_security:
 bad_fork_cleanup_policy:
 #ifdef CONFIG_NUMA
  mpol_free(p->mempolicy);
-bad_fork_cleanup_cpuset:
+bad_fork_cleanup_container:
 #endif
  cpuset_exit(p);
+ container_exit(p, container_callbacks_done);
  delayacct_tsk_free(p);
  if (p->binfmt)
   module_put(p->binfmt->module);

--
```