Subject: [PATCH 04/33] task containersv11 add fork exit hooks
Posted by Paul Menage on Mon, 17 Sep 2007 21:03:11 GMT
View Forum Message <> Reply to Message

Adds the necessary hooks to the fork() and exit() paths to ensure that
new children inherit their parent's cgroup assignments, and that exiting
processes release reference counts on their cgroups.

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

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

diff -puN include/linux/cgroup.h~task-cgroupsv11-add-fork-exit-hooks include/linux/cgroup.h
--- a/include/linux/cgroup.h~task-cgroupsv11-add-fork-exit-hooks
+++ a/include/linux/cgroup.h
@@ -25,6 +25,9 @@ extern int cgroup_init(void);
 extern void cgroup_init_smp(void);
 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_exit(struct task_struct *p, int run_callbacks);

 /* Per-subsystem/per-cgroup state maintained by the system. */
 struct cgroup_subsys_state {
@@ -215,6 +218,9 @@ int cgroup_path(const struct containe
 static inline int cgroup_init_early(void) { return 0; }
 static inline int cgroup_init(void) { return 0; }
 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_exit(struct task_struct *p, int callbacks) {}

 static inline void cgroup_lock(void) {}
 static inline void cgroup_unlock(void) {}
diff -puN kernel/cgroup.c~task-cgroupsv11-add-fork-exit-hooks kernel/cgroup.c
--- a/kernel/cgroup.c~task-cgroupsv11-add-fork-exit-hooks
+++ a/kernel/cgroup.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_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
+ * 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_set(struct css_set *cg)
+{
+ int i;
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++)
+  atomic_inc(&cg->subsys[i]->cgroup->count);
+}
+
+static void put_css_set(struct css_set *cg)
+{
+ int i;
+ for (i = 0; i < CGROUP_SUBSYS_COUNT; i++)
+  atomic_dec(&cg->subsys[i]->cgroup->count);
+}
+
 /*
  * There is one global cgroup mutex. We also require taking
  * task_lock() when dereferencing a task's cgroup subsys pointers.
@@ -1563,3 +1590,97 @@ int __init cgroup_init(void)
 out:
  return err;
 }
+
+/**
+ * cgroup_fork - attach newly forked task to its parents cgroup.
+ * @tsk: pointer to task_struct of forking parent process.
+ *
+ * Description: A task inherits its parent's cgroup at fork().
+ *
+ * A pointer to the shared css_set 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 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.
+ *
+ * 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();
+}
+
+/**
+ * cgroup_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 cgroup_fork_callbacks(struct task_struct *child)
+{
+ if (need_forkexit_callback) {
+  int i;
+  for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+   struct cgroup_subsys *ss = subsys[i];
+   if (ss->fork)
+    ss->fork(ss, child);
+  }
+ }
+}
+
+/**
+ * cgroup_exit - detach cgroup from exiting task
+ * @tsk: pointer to task_struct of exiting process
+ *
+ * Description: Detach cgroup from @tsk and release it.
+ *
+ * Note that cgroups marked notify_on_release force every task in
+ * them to take the global cgroup_mutex mutex when exiting.
+ * This could impact scaling on very large systems.  Be reluctant to
+ * use notify_on_release cgroups where very high task exit scaling
+ * is required on large systems.
+ *
+ * the_top_cgroup_hack:
+ *
+ *    Set the exiting tasks cgroup to the root cgroup (top_cgroup).
+ *
+ *    We call cgroup_exit() while the task is still competent to
+ *    handle notify_on_release(), then leave the task attached to the
+ *    root cgroup in each hierarchy for the remainder of its exit.
+ *
+ *    To do this properly, we would increment the reference count on
```

```
+ *    top_cgroup, and near the very end of the kernel/exit.c do_exit()
+ *    code we would add a second cgroup function call, to drop that
+ *    reference.  This would just create an unnecessary hot spot on
+ *    the top_cgroup reference count, to no avail.
+ *
+ *    Normally, holding a reference to a cgroup without bumping its
+ *    count is unsafe.   The cgroup could go away, or someone could
+ *    attach us to a different cgroup, decrementing the count on
+ *    the first cgroup that we never incremented.  But in this case,
+ *    top_cgroup 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 cgroup_exit(struct task_struct *tsk, int run_callbacks)
+{
+ int i;
+
+ if (run_callbacks && need_forkexit_callback) {
+  for (i = 0; i < CGROUP_SUBSYS_COUNT; i++) {
+   struct cgroup_subsys *ss = subsys[i];
+   if (ss->exit)
+    ss->exit(ss, tsk);
+  }
+ }
+ /* Reassign the task to the init_css_set. */
+ task_lock(tsk);
+ put_css_set(&tsk->cgroups);
+ tsk->cgroups = init_task.cgroups;
+ task_unlock(tsk);
+}
diff -puN kernel/exit.c~task-cgroupsv11-add-fork-exit-hooks kernel/exit.c
--- a/kernel/exit.c~task-cgroupsv11-add-fork-exit-hooks
+++ a/kernel/exit.c
@@ -33,6 +33,7 @@
 #include <linux/delayacct.h>
 #include <linux/freezer.h>
 #include <linux/cpuset.h>
+#include <linux/cgroup.h>
 #include <linux/syscalls.h>
 #include <linux/signal.h>
 #include <linux/posix-timers.h>
@@ -981,6 +982,7 @@ fastcall NORET_TYPE void do_exit(long co
  check_stack_usage();
  exit_thread();
  cpuset_exit(tsk);
+ cgroup_exit(tsk, 1);
  exit_keys(tsk);
```

```
  if (group_dead && tsk->signal->leader)
diff -puN kernel/fork.c~task-cgroupsv11-add-fork-exit-hooks kernel/fork.c
--- a/kernel/fork.c~task-cgroupsv11-add-fork-exit-hooks
+++ a/kernel/fork.c
@@ -30,6 +30,7 @@
 #include <linux/capability.h>
 #include <linux/cpu.h>
 #include <linux/cpuset.h>
+#include <linux/cgroup.h>
 #include <linux/security.h>
 #include <linux/swap.h>
 #include <linux/syscalls.h>
@@ -967,6 +968,7 @@ static struct task_struct *copy_process(
 {
  int retval;
  struct task_struct *p = NULL;
+ int cgroup_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_context = NULL;
  p->audit_context = NULL;
  cpuset_fork(p);
+ cgroup_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_cgroup;
   }
  mpol_fix_fork_child_flag(p);
 #endif
@@ -1184,6 +1187,12 @@ 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 cgroup callbacks if
+  * necessary. We need to run them before the task is visible
+  * on the tasklist. */
+ cgroup_fork_callbacks(p);
+ cgroup_callbacks_done = 1;
+
  /* Need tasklist lock for parent etc handling! */
  write_lock_irq(&tasklist_lock);
```

```
@@ -1306,9 +1315,10 @@ bad_fork_cleanup_security:
 bad_fork_cleanup_policy:
 #ifdef CONFIG_NUMA
  mpol_free(p->mempolicy);
-bad_fork_cleanup_cpuset:
+bad_fork_cleanup_cgroup:
 #endif
  cpuset_exit(p);
+ cgroup_exit(p, cgroup_callbacks_done);
  delayacct_tsk_free(p);
  if (p->binfmt)
   module_put(p->binfmt->module);
```

_

--

_____

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