Subject: [PATCH] cgroup: Remove call to synchronize_rcu in cgroup_attach_task
Posted by Colin Cross on Wed, 24 Nov 2010 01:43:38 GMT
View Forum Message <> Reply to Message

synchronize_rcu can be very expensive, averaging 100 ms in
some cases. In cgroup_attach_task, it is used to prevent
a task->cgroups pointer dereferenced in an RCU read side
critical section from being invalidated by delaying the call
to put_css_set until after an RCU grace period.

To avoid the call to synchronize_rcu, make the put_css_set
call rcu-safe by moving the deletion of the css_set links
into rcu-protected free_css_set_rcu.

The calls to check_for_release in free_css_set_rcu now occur
in softirq context, so convert all uses of the
release_list_lock spinlock to irq safe versions.

The decrement of the cgroup refcount is no longer
synchronous with the call to put_css_set, which can result
in the cgroup refcount staying positive after the last call
to cgroup_attach_task returns. To allow the cgroup to be
deleted with cgroup_rmdir synchronously after
cgroup_attach_task, introduce a second refcount,
rmdir_count, that is decremented synchronously in
put_css_set. If cgroup_rmdir is called on a cgroup for
hich rmdir_count is zero but count is nonzero, reuse the
rmdir waitqueue to block the rmdir until the rcu callback
is called.

Signed-off-by: Colin Cross <ccross@android.com>

This patch is similar to what you described. The main differences are
that I used a new atomic to handle the rmdir case, and I converted
check_for_release to be callable in softirq context rather than schedule
work in free_css_set_rcu. Your css_set scanning in rmdir sounds better,
I'll take another look at that. Is there any problem with disabling irqs
with spin_lock_irqsave in check_for_release?

```
 include/linux/cgroup.h |    6 ++
 kernel/cgroup.c        |  124 +++++++++++++++++++++++++++++++++-----------------------
 2 files changed, 78 insertions(+), 52 deletions(-)
```

diff --git a/include/linux/cgroup.h b/include/linux/cgroup.h
index ed4ba11..3b6e73d 100644
--- a/include/linux/cgroup.h
+++ b/include/linux/cgroup.h

```
@@ -202,6 +202,12 @@ struct cgroup {
  atomic_t count;

  /*
+  * separate refcount for rmdir on a cgroup.  When rmdir_count is 0,
+  * rmdir should block until count is 0.
+  */
+ atomic_t rmdir_count;
+
+ /*
   * We link our 'sibling' struct into our parent's 'children'.
   * Our children link their 'sibling' into our 'children'.
   */
diff --git a/kernel/cgroup.c b/kernel/cgroup.c
index 66a416b..fa3c0ac 100644
--- a/kernel/cgroup.c
+++ b/kernel/cgroup.c
@@ -267,6 +267,33 @@ static void cgroup_release_agent(struct work_struct *work);
 static DECLARE_WORK(release_agent_work, cgroup_release_agent);
 static void check_for_release(struct cgroup *cgrp);

+/*
+ * A queue for waiters to do rmdir() cgroup. A tasks will sleep when
+ * cgroup->count == 0 && list_empty(&cgroup->children) && subsys has some
+ * reference to css->refcnt. In general, this refcnt is expected to goes down
+ * to zero, soon.
+ *
+ * CGRP_WAIT_ON_RMDIR flag is set under cgroup's inode->i_mutex;
+ */
+DECLARE_WAIT_QUEUE_HEAD(cgroup_rmdir_waitq);
+
+static void cgroup_wakeup_rmdir_waiter(struct cgroup *cgrp)
+{
+ if (unlikely(test_and_clear_bit(CGRP_WAIT_ON_RMDIR, &cgrp->flags)))
+  wake_up_all(&cgroup_rmdir_waitq);
+}
+
+void cgroup_exclude_rmdir(struct cgroup_subsys_state *css)
+{
+ css_get(css);
+}
+
+void cgroup_release_and_wakeup_rmdir(struct cgroup_subsys_state *css)
+{
+ cgroup_wakeup_rmdir_waiter(css->cgroup);
+ css_put(css);
+}
+
```

```
 /* Link structure for associating css_set objects with cgroups */
 struct cg_cgroup_link {
  /*
@@ -329,6 +356,22 @@ static struct hlist_head *css_set_hash(struct cgroup_subsys_state
*css[])
 static void free_css_set_rcu(struct rcu_head *obj)
 {
  struct css_set *cg = container_of(obj, struct css_set, rcu_head);
+ struct cg_cgroup_link *link;
+ struct cg_cgroup_link *saved_link;
+
+ /* Nothing else can have a reference to cg, no need for css_set_lock */
+ list_for_each_entry_safe(link, saved_link, &cg->cg_links,
+     cg_link_list) {
+  struct cgroup *cgrp = link->cgrp;
+  list_del(&link->cg_link_list);
+  list_del(&link->cgrp_link_list);
+  if (atomic_dec_and_test(&cgrp->count)) {
+   check_for_release(cgrp);
+   cgroup_wakeup_rmdir_waiter(cgrp);
+  }
+  kfree(link);
+ }
+
  kfree(cg);
 }

@@ -355,23 +398,20 @@ static void __put_css_set(struct css_set *cg, int taskexit)
   return;
  }

- /* This css_set is dead. unlink it and release cgroup refcounts */
  hlist_del(&cg->hlist);
  css_set_count--;

+ /* This css_set is now unreachable from the css_set_table, but RCU
+  * read-side critical sections may still have a reference to it.
+  * Decrement the cgroup rmdir_count so that rmdir's on an empty
+  * cgroup can block until the free_css_set_rcu callback */
  list_for_each_entry_safe(link, saved_link, &cg->cg_links,
      cg_link_list) {
   struct cgroup *cgrp = link->cgrp;
-  list_del(&link->cg_link_list);
-  list_del(&link->cgrp_link_list);
-  if (atomic_dec_and_test(&cgrp->count) &&
-      notify_on_release(cgrp)) {
-   if (taskexit)
-    set_bit(CGRP_RELEASABLE, &cgrp->flags);
```

```
-   check_for_release(cgrp);
-  }
-
-  kfree(link);
+   if (taskexit)
+    set_bit(CGRP_RELEASABLE, &cgrp->flags);
+   atomic_dec(&cgrp->rmdir_count);
+   smp_mb();
   }

   write_unlock(&css_set_lock);
@@ -571,6 +611,8 @@ static void link_css_set(struct list_head *tmp_cg_links,
      cgrp_link_list);
   link->cg = cg;
   link->cgrp = cgrp;
+  atomic_inc(&cgrp->rmdir_count);
+  smp_mb(); /* make sure rmdir_count increments first */
   atomic_inc(&cgrp->count);
   list_move(&link->cgrp_link_list, &cgrp->css_sets);
   /*
@@ -725,9 +767,9 @@ static struct cgroup *task_cgroup_from_root(struct task_struct *task,
   * cgroup_attach_task(), which overwrites one tasks cgroup pointer with
   * another.  It does so using cgroup_mutex, however there are
   * several performance critical places that need to reference
-  * task->cgroup without the expense of grabbing a system global
+  * task->cgroups without the expense of grabbing a system global
   * mutex.  Therefore except as noted below, when dereferencing or, as
-  * in cgroup_attach_task(), modifying a task'ss cgroup pointer we use
+  * in cgroup_attach_task(), modifying a task's cgroups pointer we use
   * task_lock(), which acts on a spinlock (task->alloc_lock) already in
   * the task_struct routinely used for such matters.
   *
@@ -909,33 +951,6 @@ static void cgroup_d_remove_dir(struct dentry *dentry)
  }

  /*
-  * A queue for waiters to do rmdir() cgroup. A tasks will sleep when
-  * cgroup->count == 0 && list_empty(&cgroup->children) && subsys has some
-  * reference to css->refcnt. In general, this refcnt is expected to goes down
-  * to zero, soon.
-  *
-  * CGRP_WAIT_ON_RMDIR flag is set under cgroup's inode->i_mutex;
-  */
-DECLARE_WAIT_QUEUE_HEAD(cgroup_rmdir_waitq);
-
-static void cgroup_wakeup_rmdir_waiter(struct cgroup *cgrp)
-{
-  if (unlikely(test_and_clear_bit(CGRP_WAIT_ON_RMDIR, &cgrp->flags)))
```

```
-  wake_up_all(&cgroup_rmdir_waitq);
-}
-
-void cgroup_exclude_rmdir(struct cgroup_subsys_state *css)
-{
- css_get(css);
-}
-
-void cgroup_release_and_wakeup_rmdir(struct cgroup_subsys_state *css)
-{
- cgroup_wakeup_rmdir_waiter(css->cgroup);
- css_put(css);
-}
-
-/*
  * Call with cgroup_mutex held. Drops reference counts on modules, including
  * any duplicate ones that parse_cgroupfs_options took. If this function
  * returns an error, no reference counts are touched.
@@ -1802,7 +1817,7 @@ int cgroup_attach_task(struct cgroup *cgrp, struct task_struct *tsk)
    ss->attach(ss, cgrp, oldcgrp, tsk, false);
  }
  set_bit(CGRP_RELEASABLE, &oldcgrp->flags);
- synchronize_rcu();
+ /* put_css_set will not destroy cg until after an RCU grace period */
  put_css_set(cg);

  /*
@@ -3566,11 +3581,12 @@ static int cgroup_rmdir(struct inode *unused_dir, struct dentry
*dentry)
  DEFINE_WAIT(wait);
  struct cgroup_event *event, *tmp;
  int ret;
+ unsigned long flags;

  /* the vfs holds both inode->i_mutex already */
 again:
  mutex_lock(&cgroup_mutex);
- if (atomic_read(&cgrp->count) != 0) {
+ if (atomic_read(&cgrp->rmdir_count) != 0) {
   mutex_unlock(&cgroup_mutex);
   return -EBUSY;
  }
@@ -3603,13 +3619,13 @@ again:

  mutex_lock(&cgroup_mutex);
  parent = cgrp->parent;
- if (atomic_read(&cgrp->count) || !list_empty(&cgrp->children)) {
+ if (atomic_read(&cgrp->rmdir_count) || !list_empty(&cgrp->children)) {
```

```
   clear_bit(CGRP_WAIT_ON_RMDIR, &cgrp->flags);
   mutex_unlock(&cgroup_mutex);
   return -EBUSY;
  }
  prepare_to_wait(&cgroup_rmdir_waitq, &wait, TASK_INTERRUPTIBLE);
- if (!cgroup_clear_css_refs(cgrp)) {
+ if (atomic_read(&cgrp->count) != 0 || !cgroup_clear_css_refs(cgrp)) {
   mutex_unlock(&cgroup_mutex);
   /*
    * Because someone may call cgroup_wakeup_rmdir_waiter() before
@@ -3627,11 +3643,11 @@ again:
  finish_wait(&cgroup_rmdir_waitq, &wait);
  clear_bit(CGRP_WAIT_ON_RMDIR, &cgrp->flags);

- spin_lock(&release_list_lock);
+ spin_lock_irqsave(&release_list_lock, flags);
  set_bit(CGRP_REMOVED, &cgrp->flags);
  if (!list_empty(&cgrp->release_list))
   list_del(&cgrp->release_list);
- spin_unlock(&release_list_lock);
+ spin_unlock_irqrestore(&release_list_lock, flags);

  cgroup_lock_hierarchy(cgrp->root);
  /* delete this cgroup from parent->children */
@@ -4389,6 +4405,8 @@ int cgroup_is_descendant(const struct cgroup *cgrp, struct task_struct
*task)

 static void check_for_release(struct cgroup *cgrp)
 {
+ unsigned long flags;
+
  /* All of these checks rely on RCU to keep the cgroup
   * structure alive */
  if (cgroup_is_releasable(cgrp) && !atomic_read(&cgrp->count)
@@ -4397,13 +4415,13 @@ static void check_for_release(struct cgroup *cgrp)
    * already queued for a userspace notification, queue
    * it now */
   int need_schedule_work = 0;
-  spin_lock(&release_list_lock);
+  spin_lock_irqsave(&release_list_lock, flags);
   if (!cgroup_is_removed(cgrp) &&
       list_empty(&cgrp->release_list)) {
    list_add(&cgrp->release_list, &release_list);
    need_schedule_work = 1;
   }
-  spin_unlock(&release_list_lock);
+  spin_unlock_irqrestore(&release_list_lock, flags);
   if (need_schedule_work)
```

```
    schedule_work(&release_agent_work);
  }
@@ -4453,9 +4471,11 @@ EXPORT_SYMBOL_GPL(__css_put);
  */
 static void cgroup_release_agent(struct work_struct *work)
 {
+ unsigned long flags;
+
   BUG_ON(work != &release_agent_work);
   mutex_lock(&cgroup_mutex);
- spin_lock(&release_list_lock);
+ spin_lock_irqsave(&release_list_lock, flags);
   while (!list_empty(&release_list)) {
    char *argv[3], *envp[3];
    int i;
@@ -4464,7 +4484,7 @@ static void cgroup_release_agent(struct work_struct *work)
            struct cgroup,
            release_list);
    list_del_init(&cgrp->release_list);
-   spin_unlock(&release_list_lock);
+   spin_unlock_irqrestore(&release_list_lock, flags);
    pathbuf = kmalloc(PAGE_SIZE, GFP_KERNEL);
    if (!pathbuf)
     goto continue_free;
@@ -4494,9 +4514,9 @@ static void cgroup_release_agent(struct work_struct *work)
   continue_free:
    kfree(pathbuf);
    kfree(agentbuf);
-   spin_lock(&release_list_lock);
+   spin_lock_irqsave(&release_list_lock, flags);
   }
- spin_unlock(&release_list_lock);
+ spin_unlock_irqrestore(&release_list_lock, flags);
   mutex_unlock(&cgroup_mutex);
 }

--
1.7.3.1
```

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containe rs