Subject: [PATCH v4 12/14] execute the whole memcg freeing in free_worker
Posted by Glauber Costa on Mon, 08 Oct 2012 10:06:18 GMT
View Forum Message <> Reply to Message

A lot of the initialization we do in mem_cgroup_create() is done with softirqs enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This means that the freeing of memcg structure must happen in a compatible context, otherwise we'll get a deadlock, like the one bellow, caught by lockdep:

```
 [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
 [<ffffffff81047f90>] free_task+0x31/0x5c
 [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
 [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
 [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
 [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
 [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
 [<ffffffff8105266d>] __do_softirq+0x122/0x277
```

This usage pattern could not be triggered before kmem came into play. With the introduction of kmem stack handling, it is possible that we call the last mem_cgroup_put() from the task destructor, which is run in an rcu callback. Such callbacks are run with softirqs disabled, leading to the offensive usage pattern.

In general, we have little, if any, means to guarantee in which context the last memcg_put will happen. The best we can do is test it and try to make sure no invalid context releases are happening. But as we add more code to memcg, the possible interactions grow in number and expose more ways to get context conflicts. One thing to keep in mind, is that part of the freeing process is already deferred to a worker, such as vfree(), that can only be called from process context.

For the moment, the only two functions we really need moved away are:

 * free_css_id(), and
 * mem_cgroup_remove_from_trees().

But because the later accesses per-zone info, free_mem_cgroup_per_zone_info() needs to be moved as well. With that, we are left with the per_cpu stats only. Better move it all.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Tested-by: Greg Thelen <gthelen@google.com>
CC: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Michal Hocko <mhocko@suse.cz>

CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Tejun Heo <tj@kernel.org>

```
---
 mm/memcontrol.c | 66 ++++++++++++++++++++++++++++++++++--------------------------------
 1 file changed, 34 insertions(+), 32 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2f92f89..c5215f1 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5205,16 +5205,29 @@ out_free:
 }

 /*
- * Helpers for freeing a kmalloc()ed/vzalloc()ed mem_cgroup by RCU,
- * but in process context.  The work_freeing structure is overlaid
- * on the rcu_freeing structure, which itself is overlaid on memsw.
+ * At destroying mem_cgroup, references from swap_cgroup can remain.
+ * (scanning all at force_empty is too costly...)
+ *
+ * Instead of clearing all references at force_empty, we remember
+ * the number of reference from swap_cgroup and free mem_cgroup when
+ * it goes down to 0.
+ *
+ * Removal of cgroup itself succeeds regardless of refs from swap.
  */
-static void free_work(struct work_struct *work)
+
+static void __mem_cgroup_free(struct mem_cgroup *memcg)
 {
- struct mem_cgroup *memcg;
+ int node;
  int size = sizeof(struct mem_cgroup);

- memcg = container_of(work, struct mem_cgroup, work_freeing);
+ mem_cgroup_remove_from_trees(memcg);
+ free_css_id(&mem_cgroup_subsys, &memcg->css);
+
+ for_each_node(node)
+  free_mem_cgroup_per_zone_info(memcg, node);
+
+ free_percpu(memcg->stat);
+
  /*
   * We need to make sure that (at least for now), the jump label
   * destruction code runs outside of the cgroup lock. This is because
@@ -5233,38 +5246,27 @@ static void free_work(struct work_struct *work)
   vfree(memcg);
 }

-static void free_rcu(struct rcu_head *rcu_head)
-{
- struct mem_cgroup *memcg;
-
- memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
- INIT_WORK(&memcg->work_freeing, free_work);
- schedule_work(&memcg->work_freeing);
-}

 /*
- * At destroying mem_cgroup, references from swap_cgroup can remain.
- * (scanning all at force_empty is too costly...)
- *
- * Instead of clearing all references at force_empty, we remember
- * the number of reference from swap_cgroup and free mem_cgroup when
- * it goes down to 0.
- *
- * Removal of cgroup itself succeeds regardless of refs from swap.
+ * Helpers for freeing a kmalloc()ed/vzalloc()ed mem_cgroup by RCU,
+ * but in process context.  The work_freeing structure is overlaid
+ * on the rcu_freeing structure, which itself is overlaid on memsw.
  */
-
-static void __mem_cgroup_free(struct mem_cgroup *memcg)
+static void free_work(struct work_struct *work)
 {
- int node;
+ struct mem_cgroup *memcg;

- mem_cgroup_remove_from_trees(memcg);
- free_css_id(&mem_cgroup_subsys, &memcg->css);
+ memcg = container_of(work, struct mem_cgroup, work_freeing);
+ __mem_cgroup_free(memcg);
+}

- for_each_node(node)
-  free_mem_cgroup_per_zone_info(memcg, node);
+static void free_rcu(struct rcu_head *rcu_head)
+{
+ struct mem_cgroup *memcg;

- free_percpu(memcg->stat);
- call_rcu(&memcg->rcu_freeing, free_rcu);
+ memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
+ INIT_WORK(&memcg->work_freeing, free_work);
+ schedule_work(&memcg->work_freeing);
```

```
 }

 static void mem_cgroup_get(struct mem_cgroup *memcg)
@@ -5276,7 +5278,7 @@ static void __mem_cgroup_put(struct mem_cgroup *memcg, int count)
 {
  if (atomic_sub_and_test(count, &memcg->refcnt)) {
   struct mem_cgroup *parent = parent_mem_cgroup(memcg);
-  __mem_cgroup_free(memcg);
+  call_rcu(&memcg->rcu_freeing, free_rcu);
   if (parent)
    mem_cgroup_put(parent);
  }
--
1.7.11.4
```