Subject: [PATCH v5 14/18] memcg/sl[au]b: shrink dead caches
Posted by Glauber Costa on Fri, 19 Oct 2012 14:20:38 GMT
View Forum Message <> Reply to Message

In the slub allocator, when the last object of a page goes away, we
don't necessarily free it - there is not necessarily a test for empty
page in any slab_free path.

This means that when we destroy a memcg cache that happened to be empty,
those caches may take a lot of time to go away: removing the memcg
reference won't destroy them - because there are pending references, and
the empty pages will stay there, until a shrinker is called upon for any
reason.

This patch marks all memcg caches as dead. kmem_cache_shrink is called
for the ones who are not yet dead - this will force internal cache
reorganization, and then all references to empty pages will be removed.

An unlikely branch is used to make sure this case does not affect
performance in the usual slab_free path.

The slab allocator has a time based reaper that would eventually get rid
of the objects, but we can also call it explicitly, since dead caches
are not a likely event.

[v2: also call verify_dead for the slab]
[v3: use delayed_work to avoid calling verify_dead at every free]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Tejun Heo <tj@kernel.org>

```
---
 include/linux/slab.h |  2 +-
 mm/memcontrol.c      | 47 ++++++++++++++++++++++++++++++++++++++++++------
 2 files changed, 42 insertions(+), 7 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index bb698dc..4a3a749 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -212,7 +212,7 @@ struct memcg_cache_params {
    struct kmem_cache *root_cache;
    bool dead;
```

```
    atomic_t nr_pages;
-   struct work_struct destroy;
+   struct delayed_work destroy;
   };
  };
 };
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index f5089b3..c7732fa 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2956,14 +2956,37 @@ static void kmem_cache_destroy_work_func(struct work_struct *w)
 {
  struct kmem_cache *cachep;
  struct memcg_cache_params *p;
+ struct delayed_work *dw = to_delayed_work(w);

- p = container_of(w, struct memcg_cache_params, destroy);
+ p = container_of(dw, struct memcg_cache_params, destroy);

  VM_BUG_ON(p->is_root_cache);
  cachep = p->root_cache;
  cachep = cachep->memcg_params->memcg_caches[memcg_css_id(p->memcg)];

- if (!atomic_read(&cachep->memcg_params->nr_pages))
+ /*
+  * If we get down to 0 after shrink, we could delete right away.
+  * However, memcg_release_pages() already puts us back in the workqueue
+  * in that case. If we proceed deleting, we'll get a dangling
+  * reference, and removing the object from the workqueue in that case
+  * is unnecessary complication. We are not a fast path.
+  *
+  * Note that this case is fundamentally different from racing with
+  * shrink_slab(): if memcg_cgroup_destroy_cache() is called in
+  * kmem_cache_shrink, not only we would be reinserting a dead cache
+  * into the queue, but doing so from inside the worker racing to
+  * destroy it.
+  *
+  * So if we aren't down to zero, we'll just schedule a worker and try
+  * again
+  */
+ if (atomic_read(&cachep->memcg_params->nr_pages) != 0) {
+  kmem_cache_shrink(cachep);
+  if (atomic_read(&cachep->memcg_params->nr_pages) == 0)
+   return;
+  /* Once per minute should be good enough. */
+  schedule_delayed_work(&cachep->memcg_params->destroy, 60 * HZ);
+ } else
   kmem_cache_destroy(cachep);
```

```
 }

@@ -2973,10 +2996,22 @@ void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
   return;

  /*
+  * We can get to a memory-pressure situation while the delayed work is
+  * still pending to run. The vmscan shrinkers can then release all
+  * cache memory and get us to destruction. If this is the case, we'll
+  * be executed twice, which is a bug (the second time will execute over
+  * bogus data).
+  *
+  * Since we can't possibly know who got us here, just refrain from
+  * running if there is already work pending
+  */
+ if (delayed_work_pending(&cachep->memcg_params->destroy))
+  return;
+ /*
   * We have to defer the actual destroying to a workqueue, because
   * we might currently be in a context that cannot sleep.
   */
- schedule_work(&cachep->memcg_params->destroy);
+ schedule_delayed_work(&cachep->memcg_params->destroy, 0);
 }

 /*
@@ -3142,9 +3177,9 @@ static void mem_cgroup_destroy_all_caches(struct mem_cgroup
*memcg)
  list_for_each_entry(cachep, &memcg->memcg_slab_caches, list) {

   cachep->memcg_params->dead = true;
-  INIT_WORK(&cachep->memcg_params->destroy,
-    kmem_cache_destroy_work_func);
-  schedule_work(&cachep->memcg_params->destroy);
+  INIT_DELAYED_WORK(&cachep->memcg_params->destroy,
+      kmem_cache_destroy_work_func);
+  schedule_delayed_work(&cachep->memcg_params->destroy, 0);
  }
  mutex_unlock(&memcg->slab_caches_mutex);
 }
--
1.7.11.7
```