Subject: [PATCH 07/10] memcg: destroy memcg caches
Posted by Glauber Costa on Wed, 25 Jul 2012 14:38:18 GMT
View Forum Message <> Reply to Message

This patch implements destruction of memcg caches. Right now,
only caches where our reference counter is the last remaining are
deleted. If there are any other reference counters around, we just
leave the caches lying around until they go away.

When that happen, a destruction function is called from the cache
code. Caches are only destroyed in process context, so we queue them
up for later processing in the general case.

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>
---
 include/linux/memcontrol.h | 1 +
 include/linux/slab.h | 3 ++
 mm/memcontrol.c | 84 ++
 mm/slab.c | 4 +++
 mm/slab.h | 21 ++++++++++
 mm/slub.c | 7 +++-
 6 files changed, 119 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index bd1f34b..247019f 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -425,6 +425,7 @@ void memcg_register_cache(struct mem_cgroup *memcg,
 void memcg_release_cache(struct kmem_cache *cachep);
 struct kmem_cache *
 __memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
 #else
 static inline void memcg_register_cache(struct mem_cgroup *memcg,
                                         struct kmem_cache *s)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 249a0d3..b9310a4 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -186,6 +186,9 @@ struct mem_cgroup_cache_params {
   struct mem_cgroup *memcg;
   struct kmem_cache *parent;

```
  int id;
+ bool dead;
+ atomic_t nr_pages;
+ struct list_head destroyed_list; /* Used when deleting memcg cache */
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2cc3acf..1231d86 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -599,6 +599,8 @@ void memcg_register_cache(struct mem_cgroup *memcg, struct
kmem_cache *cachep)
 {
  int id = -1;

+ INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
+
  if (!memcg)
   id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
         GFP_KERNEL);
@@ -768,6 +770,7 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  mem_cgroup_get(memcg);
  memcg->slabs[idx] = new_cachep;
  new_cachep->memcg_params.memcg = memcg;
+ atomic_set(&new_cachep->memcg_params.nr_pages , 0);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -782,6 +785,55 @@ struct create_work {
 /* Use a single spinlock for destruction and creation, not a frequent op */
 static DEFINE_SPINLOCK(cache_queue_lock);
 static LIST_HEAD(create_queue);
+static LIST_HEAD(destroyed_caches);
+
+static void kmem_cache_destroy_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ struct mem_cgroup_cache_params *p, *tmp;
+ unsigned long flags;
+ LIST_HEAD(del_unlocked);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_move(&cachep->memcg_params.destroyed_list, &del_unlocked);
+ }
```

```
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(p, tmp, &del_unlocked, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_del(&cachep->memcg_params.destroyed_list);
+  if (!atomic_read(&cachep->memcg_params.nr_pages)) {
+   mem_cgroup_put(cachep->memcg_params.memcg);
+   kmem_cache_destroy(cachep);
+  }
+ }
+}
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
+static void __mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ BUG_ON(cachep->memcg_params.id != -1);
+ list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
+}
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ unsigned long flags;
+
+ if (!cachep->memcg_params.dead)
+  return;
+ /*
+  * We have to defer the actual destroying to a workqueue, because
+  * we might currently be in a context that cannot sleep.
+  */
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ __mem_cgroup_destroy_cache(cachep);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}

 /*
  * Flush the queue of kmem_caches to create, because we're creating a cgroup.
@@ -803,6 +855,33 @@ void memcg_flush_cache_create_queue(void)
  spin_unlock_irqrestore(&cache_queue_lock, flags);
 }

+static void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+ struct kmem_cache *cachep;
+ unsigned long flags;
+ int i;
+
```

```
+ /*
+  * pre_destroy() gets called with no tasks in the cgroup.
+  * this means that after flushing the create queue, no more caches
+  * will appear
+  */
+ memcg_flush_cache_create_queue();
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  cachep = memcg->slabs[i];
+  if (!cachep)
+   continue;
+
+  cachep->memcg_params.dead = true;
+  __mem_cgroup_destroy_cache(cachep);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}
+
 static void memcg_create_cache_work_func(struct work_struct *w)
 {
  struct create_work *cw, *tmp;
@@ -914,6 +993,10 @@ EXPORT_SYMBOL(__memcg_kmem_get_cache);
 static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
 }
+
+static inline void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */

 #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
@@ -5517,6 +5600,7 @@ static int mem_cgroup_pre_destroy(struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);

+ mem_cgroup_destroy_all_caches(memcg);
  return mem_cgroup_force_empty(memcg, false);
 }

diff --git a/mm/slab.c b/mm/slab.c
index ddc60a4..21d7cf7 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1769,6 +1769,8 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
```

```
int nodeid)
  for (i = 0; i < nr_pages; i++)
   __SetPageSlab(page + i);

+ mem_cgroup_bind_pages(cachep, cachep->gfporder);
+
  if (kmemcheck_enabled && !(cachep->flags & SLAB_NOTRACK)) {
   kmemcheck_alloc_shadow(page, cachep->gfporder, flags, nodeid);

@@ -1803,6 +1805,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
   __ClearPageSlab(page);
   page++;
  }
+
+ mem_cgroup_release_pages(cachep, cachep->gfporder);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += nr_freed;
  free_pages((unsigned long)addr, cachep->gfporder);
diff --git a/mm/slab.h b/mm/slab.h
index 3c637d2..d9df178 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -1,5 +1,6 @@
 #ifndef MM_SLAB_H
 #define MM_SLAB_H
+#include <linux/memcontrol.h>
 /*
  * Internal slab definitions
  */
@@ -66,6 +67,19 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
        return cachep->memcg_params.memcg == memcg;
 }

+static inline void mem_cgroup_bind_pages(struct kmem_cache *s, int order)
+{
+ if (s->memcg_params.id == -1)
+  atomic_add(1 << order, &s->memcg_params.nr_pages);
+}
+
+static inline void mem_cgroup_release_pages(struct kmem_cache *s, int order)
+{
+ if (s->memcg_params.id != -1)
+  return;
+ if (atomic_sub_and_test((1 << order), &s->memcg_params.nr_pages))
+  mem_cgroup_destroy_cache(s);
+}
 #else
 static inline bool slab_is_parent(struct kmem_cache *s, struct kmem_cache *p)
```

```
 {
@@ -77,4 +91,11 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
 {
        return true;
 }
+
+static inline void mem_cgroup_bind_pages(struct kmem_cache *s, int order)
+{
+}
+static inline void mem_cgroup_release_pages(struct kmem_cache *s, int order)
+{
+}
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index 6175a72..fdb1a0d 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1344,6 +1344,7 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  void *start;
  void *last;
  void *p;
+ int order;

  BUG_ON(flags & GFP_SLAB_BUG_MASK);

@@ -1352,14 +1353,16 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  if (!page)
   goto out;

+ order = compound_order(page);
  inc_slabs_node(s, page_to_nid(page), page->objects);
+ mem_cgroup_bind_pages(s, order);
  page->slab = s;
  __SetPageSlab(page);

  start = page_address(page);

  if (unlikely(s->flags & SLAB_POISON))
-  memset(start, POISON_INUSE, PAGE_SIZE << compound_order(page));
+  memset(start, POISON_INUSE, PAGE_SIZE << order);

  last = start;
  for_each_object(p, s, start, page->objects) {
@@ -1399,6 +1402,8 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
   -pages);
```

```
  __ClearPageSlab(page);
+
+ mem_cgroup_release_pages(s, order);
  reset_page_mapcount(page);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
--
1.7.10.4
```