Subject: [PATCH v4 20/25] memcg: destroy memcg caches
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:13 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 | 2 +
 include/linux/slab.h | 4 +-
 mm/memcontrol.c | 95 ++-
 mm/slab.c | 4 ++
 mm/slab.h | 24 ++++++++++
 mm/slub.c | 6 ++-
 6 files changed, 131 insertions(+), 4 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 47ccd80..8ebbcc9 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -462,6 +462,8 @@ static inline bool has_memcg_flag(gfp_t gfp)

 extern struct static_key mem_cgroup_kmem_enabled_key;
 #define mem_cgroup_kmem_on static_key_false(&mem_cgroup_kmem_enabled_key)
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *s)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index d2d2fad..fb4fb7b 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -186,7 +186,9 @@ struct mem_cgroup_cache_params {
  struct mem_cgroup *memcg;

```
  struct kmem_cache *parent;
  int id;
- atomic_t refcnt;
+ 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 5295ab6..e0b79f0 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -476,6 +476,11 @@ static void disarm_static_keys(struct mem_cgroup *memcg)
 {
  if (memcg->kmem_accounted)
   static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
+ /*
+  * This check can't live in kmem destruction function,
+  * since the charges will outlive the cgroup
+  */
+ BUG_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }

 #ifdef CONFIG_INET
@@ -555,6 +560,8 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
 {
  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);
@@ -595,7 +602,7 @@ 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.refcnt, 1);
+ atomic_set(&new_cachep->memcg_params.nr_pages , 0);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -610,6 +617,55 @@ 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.
@@ -631,6 +687,33 @@ void mem_cgroup_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
+  */
+ mem_cgroup_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;
@@ -845,6 +928,10 @@ static void memcg_slab_init(struct mem_cgroup *memcg)
 static inline void disarm_static_keys(struct mem_cgroup *memcg)
 {
 }
+
+static inline void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
```

```
 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4118,6 +4205,7 @@ static int mem_cgroup_force_empty(struct mem_cgroup *memcg, bool
free_all)
  int node, zid, shrink;
  int nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
  struct cgroup *cgrp = memcg->css.cgroup;
+ u64 usage;

  css_get(&memcg->css);

@@ -4157,8 +4245,10 @@ move_account:
   if (ret == -ENOMEM)
    goto try_to_free;
   cond_resched();
+  usage = res_counter_read_u64(&memcg->res, RES_USAGE) -
+   res_counter_read_u64(&memcg->kmem, RES_USAGE);
  /* "ret" should also be checked to ensure all lists are empty. */
- } while (res_counter_read_u64(&memcg->res, RES_USAGE) > 0 || ret);
+ } while (usage > 0 || ret);
 out:
  css_put(&memcg->css);
  return ret;
@@ -5511,6 +5601,7 @@ 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 3da5210..4951c81 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1766,6 +1766,8 @@ 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);

@@ -1795,6 +1797,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
  else
   sub_zone_page_state(page_zone(page),
     NR_SLAB_UNRECLAIMABLE, nr_freed);
```

```
+
+ 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 0a3e712..0b0cab4 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
  */
@@ -73,4 +74,27 @@ static inline bool slab_is_parent(struct kmem_cache *s,
  return false;
 #endif
 }
+
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
+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 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
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index 77944e2..d5b91f4 100644
--- a/mm/slub.c
+++ b/mm/slub.c
```

```
@@ -1345,6 +1345,7 @@ 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);

@@ -1353,14 +1354,16 @@ 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;
  page->flags |= 1 << PG_slab;

  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 +1402,7 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
   NR_SLAB_RECLAIMABLE : NR_SLAB_UNRECLAIMABLE,
   -pages);

+ mem_cgroup_release_pages(s, order);
  reset_page_mapcount(page);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
--
1.7.10.2
```