Subject: [PATCH 4/7] chained slab caches: move pages to a different cache when a cache is destroyed.
Posted by Glauber Costa on Tue, 21 Feb 2012 11:34:36 GMT
View Forum Message <> Reply to Message

In the context of tracking kernel memory objects to a cgroup, the
following problem appears: we may need to destroy a cgroup, but
this does not guarantee that all objects inside the cache are dead.
This can't be guaranteed even if we shrink the cache beforehand.

The simple option is to simply leave the cache around. However,
intensive workloads may have generated a lot of objects and thus
the dead cache will live in memory for a long while.

Scanning the list of objects in the dead cache takes time, and
would probably require us to lock the free path of every objects
to make sure we're not racing against the update.

I decided to give a try to a different idea then - but I'd be
happy to pursue something else if you believe it would be better.

Upon memcg destruction, all the pages on the partial list
are moved to the new slab (usually the parent memcg, or root memcg)
When an object is freed, there are high stakes that no list locks
are needed - so this case poses no overhead. If list manipulation
is indeed needed, we can detect this case, and perform it
in the right slab.

If all pages were residing in the partial list, we can free
the cache right away. Otherwise, we do it when the last cache
leaves the full list.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Kirill A. Shutemov <kirill@shutemov.name>
CC: Greg Thelen <gthelen@google.com>
CC: Johannes Weiner <jweiner@redhat.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
CC: Paul Turner <pjt@google.com>
CC: Frederic Weisbecker <fweisbec@gmail.com>
CC: Pekka Enberg <penberg@kernel.org>
CC: Christoph Lameter <cl@linux.com>
---
 include/linux/memcontrol.h | 1 +
 include/linux/slab.h | 4 +
 include/linux/slub_def.h | 1 +
 mm/memcontrol.c | 64 ++++++++++++++++++-
 mm/slub.c | 171 ++----

```
 5 files changed, 227 insertions(+), 14 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 95e7e19..6138d10 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -32,6 +32,7 @@ struct mm_struct;

 struct memcg_kmem_cache {
  struct kmem_cache *cache;
+ struct work_struct destroy;
  struct mem_cgroup *memcg; /* Should be able to do without this */
 };

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 8a372cd..c181927 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -79,6 +79,9 @@
 /* The following flags affect the page allocator grouping pages by mobility */
 #define SLAB_RECLAIM_ACCOUNT 0x00020000UL  /* Objects are reclaimable */
 #define SLAB_TEMPORARY  SLAB_RECLAIM_ACCOUNT /* Objects are short-lived */
+
+#define SLAB_CHAINED  0x04000000UL /* Slab is dead, but some objects still points
+         to it.*/
 /*
  * ZERO_SIZE_PTR will be returned for zero sized kmalloc requests.
  *
@@ -113,6 +116,7 @@ void kmem_cache_destroy(struct kmem_cache *);
 int kmem_cache_shrink(struct kmem_cache *);
 void kmem_cache_free(struct kmem_cache *, void *);
 unsigned int kmem_cache_size(struct kmem_cache *);
+void kmem_switch_slab(struct kmem_cache *old, struct kmem_cache *new);

 /*
  * Please use this macro to create slab caches. Simply specify the
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 4f39fff..e20da0e 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -38,6 +38,7 @@ enum stat_item {
  CMPXCHG_DOUBLE_FAIL, /* Number of times that cmpxchg double did not match */
  CPU_PARTIAL_ALLOC, /* Used cpu partial on alloc */
  CPU_PARTIAL_FREE, /* USed cpu partial on free */
+ CPU_CHAINED_FREE, /* Chained to a parent slab during free*/
  NR_SLUB_STAT_ITEMS };

 struct kmem_cache_cpu {
```

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2aa35b0..1b1db88 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -4984,6 +4984,22 @@ err_cleanup:

 }

+static void __memcg_cache_destroy(struct memcg_kmem_cache *cache)
+{
+ if (!slab_nr_pages(cache->cache)) {
+  kmem_cache_destroy(cache->cache);
+  mem_cgroup_put(cache->memcg);
+ }
+}
+
+static void memcg_cache_destroy(struct work_struct *work)
+{
+ struct memcg_kmem_cache *cache;
+
+ cache = container_of(work, struct memcg_kmem_cache, destroy);
+ __memcg_cache_destroy(cache);
+}
+
 struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index)
 {
  return &memcg->kmem_cache[index];
@@ -5066,6 +5082,7 @@ void memcg_create_kmem_caches(struct mem_cgroup *memcg)
    memcg->kmem_cache[i].cache = cache;
   else
    memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
+  INIT_WORK(&memcg->kmem_cache[i].destroy, memcg_cache_destroy);
   memcg->kmem_cache[i].memcg = memcg;
  }
 }
@@ -5140,12 +5157,57 @@ free_out:
  return ERR_PTR(error);
 }

+void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg)
+{
+ int i;
+
+ for (i = 0; i < NR_CACHES; i++) {
+  if (memcg->kmem_cache[i].cache != cache)
+   continue;
+  schedule_work(&memcg->kmem_cache[i].destroy);
+ }
```

```
+}
+
 static int mem_cgroup_pre_destroy(struct cgroup_subsys *ss,
       struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);
+ struct kmem_cache *newcache, *oldcache;
+ int i;
+ int ret;
+
+ if (WARN_ON(mem_cgroup_is_root(memcg)))
+  goto out;
+
+ for (i = 0; i < NR_CACHES; i++) {
+  unsigned long pages;
+
+  if (!memcg->kmem_cache[i].cache)
+   continue;
+
+  if (!parent)
+   parent = root_mem_cgroup;

- return mem_cgroup_force_empty(memcg, false);
+  if (parent->use_hierarchy)
+   newcache = parent->kmem_cache[i].cache;
+  else
+   newcache = root_mem_cgroup->kmem_cache[i].cache;
+
+
+  oldcache = memcg->kmem_cache[i].cache;
+  pages = slab_nr_pages(oldcache);
+
+  if (pages) {
+   kmem_switch_slab(oldcache, newcache);
+   res_counter_uncharge_until(memcg_kmem(memcg), memcg_kmem(memcg)->parent,
+         pages << PAGE_SHIFT);
+  }
+  __memcg_cache_destroy(&memcg->kmem_cache[i]);
+ }
+out:
+ ret = mem_cgroup_force_empty(memcg, false);
+ return ret;
 }

 static void mem_cgroup_destroy(struct cgroup_subsys *ss,
diff --git a/mm/slub.c b/mm/slub.c
index f3815ec..22851d7 100644
```

```
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1425,7 +1425,7 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
  __free_pages(page, order);

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
- if (s->memcg)
+ if (s->memcg && !(s->flags & SLAB_CHAINED))
   memcg_kmem_freepage(s->memcg, page, 1 << order);
 #endif

@@ -1471,7 +1471,10 @@ static void free_slab(struct kmem_cache *s, struct page *page)

 static void discard_slab(struct kmem_cache *s, struct page *page)
 {
- dec_slabs_node(s, page_to_nid(page), page->objects);
+ if (s->flags & SLAB_CHAINED)
+  dec_slabs_node(s->parent_slab, page_to_nid(page), page->objects);
+ else
+  dec_slabs_node(s, page_to_nid(page), page->objects);
  free_slab(s, page);
 }

@@ -2412,6 +2415,40 @@ EXPORT_SYMBOL(kmem_cache_alloc_node_trace);
 #endif
 #endif

+static void move_previous_full(struct kmem_cache *s, struct page *page)
+{
+ struct kmem_cache *target;
+ struct kmem_cache_node *n;
+ struct kmem_cache_node *tnode;
+ unsigned long uninitialized_var(flags);
+ int node = page_to_nid(page);
+
+ if (WARN_ON(s->flags & SLAB_STORE_USER))
+  return;
+
+ target = s->parent_slab;
+ if (WARN_ON(!target))
+  goto out;
+
+ n = get_node(s, node);
+ tnode = get_node(target, node);
+
+ dec_slabs_node(s, node, page->objects);
+ inc_slabs_node(target, node, page->objects);
+
```

```
+ page->slab = target;
+ add_partial(tnode, page, DEACTIVATE_TO_TAIL);
+out:
+ up_write(&slub_lock);
+}
+
+void destroy_chained_slab(struct kmem_cache *s, struct page *page)
+{
+ struct kmem_cache_node *n = get_node(s, page_to_nid(page));
+ if (atomic_long_read(&n->nr_slabs) == 0)
+  memcg_slab_destroy(s, s->memcg);
+}
+
 /*
  * Slow patch handling. This may still be called frequently since objects
  * have a longer lifetime than the cpu slabs in most processing loads.
@@ -2431,12 +2468,19 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
  unsigned long counters;
  struct kmem_cache_node *n = NULL;
  unsigned long uninitialized_var(flags);
+ bool chained = s->flags & SLAB_CHAINED;

  stat(s, FREE_SLOWPATH);

  if (kmem_cache_debug(s) && !free_debug_processing(s, page, x, addr))
   return;

+ if (chained) {
+  /* Will only stall during the chaining, very unlikely */
+  do {} while (unlikely(!s->parent_slab));
+  stat(s->parent_slab, CPU_CHAINED_FREE);
+ }
+
  do {
   prior = page->freelist;
   counters = page->counters;
@@ -2455,8 +2499,10 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
     new.frozen = 1;

    else { /* Needs to be taken off a list */
-
-                        n = get_node(s, page_to_nid(page));
+    if (chained)
+     n = get_node(s->parent_slab, page_to_nid(page));
+    else
+     n = get_node(s, page_to_nid(page));
     /*
      * Speculatively acquire the list_lock.
```

```
     * If the cmpxchg does not succeed then we may
@@ -2482,8 +2528,19 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
    * If we just froze the page then put it onto the
    * per cpu partial list.
    */
-  if (new.frozen && !was_frozen)
-   put_cpu_partial(s, page, 1);
+  if (new.frozen && !was_frozen) {
+   if (!(chained))
+    put_cpu_partial(s, page, 1);
+   else  {
+    if (unlikely(page->slab == s)) {
+     n = get_node(s, page_to_nid(page));
+     spin_lock_irqsave(&n->list_lock, flags);
+     move_previous_full(s, page);
+     spin_unlock_irqrestore(&n->list_lock, flags);
+    } else
+     destroy_chained_slab(s, page);
+   }
+  }

   /*
    * The list lock was not taken therefore no list
@@ -2501,7 +2558,7 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
  if (was_frozen)
   stat(s, FREE_FROZEN);
  else {
-  if (unlikely(!inuse && n->nr_partial > s->min_partial))
+  if (unlikely(!inuse && ((n->nr_partial > s->min_partial) || (chained))))
                        goto slab_empty;

   /*
@@ -2509,10 +2566,18 @@ static void __slab_free(struct kmem_cache *s, struct page *page,
    * then add it.
    */
   if (unlikely(!prior)) {
+   struct kmem_cache *target = s;
    remove_full(s, page);
-   add_partial(n, page, DEACTIVATE_TO_TAIL);
-   stat(s, FREE_ADD_PARTIAL);
+
+   if ((page->slab == s) && (chained))
+    move_previous_full(s, page);
+   else
+    add_partial(n, page, DEACTIVATE_TO_TAIL);
+   stat(target, FREE_ADD_PARTIAL);
   }
+
```

```
+  if (page->slab != s)
+   destroy_chained_slab(s, page);
  }
  spin_unlock_irqrestore(&n->list_lock, flags);
  return;
@@ -2522,13 +2587,26 @@ slab_empty:
   /*
    * Slab on the partial list.
    */
-  remove_partial(n, page);
-  stat(s, FREE_REMOVE_PARTIAL);
- } else
+  if (likely(!chained)) {
+   remove_partial(n, page);
+   stat(s, FREE_REMOVE_PARTIAL);
+  }
+ } else {
   /* Slab must be on the full list */
   remove_full(s, page);
+  /*
+   * In chaining, an empty slab can't be in the
+   * full list. We should have removed it when the first
+   * object was freed from it
+   */
+  WARN_ON((page->slab == s) && chained);
+ }

  spin_unlock_irqrestore(&n->list_lock, flags);
+
+ if (page->slab != s)
+  destroy_chained_slab(s, page);
+
  stat(s, FREE_SLAB);
  discard_slab(s, page);
 }
@@ -2577,8 +2655,11 @@ redo:
    goto redo;
   }
   stat(s, FREE_FASTPATH);
- } else
+ } else {
+  rcu_read_lock();
   __slab_free(s, page, x, addr);
+  rcu_read_unlock();
+ }

 }
```

```
@@ -3150,6 +3231,66 @@ static void free_partial(struct kmem_cache *s, struct
kmem_cache_node *n)
  }
 }

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void kmem_switch_slab(struct kmem_cache *old, struct kmem_cache *new)
+{
+ int node;
+ unsigned long flags;
+ struct page *page, *spage;
+
+ /*
+  * We need to make sure nobody is at slab_free's slow path.
+  * everybody before that will see the slab as non-chained.
+  *
+  * After the flag is set, we don't care if new freers appear.
+  * The node list lock should be able to protect any list operation
+  * being made. And if we're chained, we'll make sure we're grabbing
+  * the lock to the target cache.
+  *
+  * We also need to make sure nobody puts a page back to the per-cpu
+  * list.
+  */
+ synchronize_rcu();
+ old->flags |= SLAB_CHAINED;
+
+ flush_all(old);
+ for_each_online_node(node) {
+  struct kmem_cache_node *nnew = get_node(new, node);
+  struct kmem_cache_node *nold = get_node(old, node);
+
+  if (!nnew)
+   continue;
+
+  WARN_ON(!nold);
+  spin_lock_irqsave(&nnew->list_lock, flags);
+
+  list_for_each_entry_safe(page, spage, &nold->partial, lru) {
+   dec_slabs_node(old, node, page->objects);
+   page->slab = new;
+   inc_slabs_node(new, node, page->objects);
+   remove_partial(nold, page);
+   add_partial(nnew, page, DEACTIVATE_TO_TAIL);
+  }
+
+  if (!(old->flags & SLAB_STORE_USER)) {
+   spin_unlock_irqrestore(&nnew->list_lock, flags);
```

```
+   continue;
+  }
+
+  list_for_each_entry_safe(page, spage, &nold->full, lru) {
+   dec_slabs_node(old, node, page->objects);
+   page->slab = new;
+   inc_slabs_node(new, node, page->objects);
+   remove_full(old, page);
+   add_full(new, nnew, page);
+  }
+  spin_unlock_irqrestore(&nnew->list_lock, flags);
+
+ }
+ old->parent_slab = new;
+}
+#endif
+
 /*
  * Release all resources used by a slab cache.
  */
@@ -5108,6 +5249,7 @@ STAT_ATTR(CMPXCHG_DOUBLE_CPU_FAIL,
cmpxchg_double_cpu_fail);
 STAT_ATTR(CMPXCHG_DOUBLE_FAIL, cmpxchg_double_fail);
 STAT_ATTR(CPU_PARTIAL_ALLOC, cpu_partial_alloc);
 STAT_ATTR(CPU_PARTIAL_FREE, cpu_partial_free);
+STAT_ATTR(CPU_CHAINED_FREE, cpu_chained_free);
 #endif

 static struct attribute *slab_attrs[] = {
@@ -5173,6 +5315,7 @@ static struct attribute *slab_attrs[] = {
  &cmpxchg_double_cpu_fail_attr.attr,
  &cpu_partial_alloc_attr.attr,
  &cpu_partial_free_attr.attr,
+ &cpu_chained_free_attr.attr,
 #endif
 #ifdef CONFIG_FAILSLAB
  &failslab_attr.attr,
@@ -5487,6 +5630,8 @@ static int s_show(struct seq_file *m, void *p)
  int node;

  s = list_entry(p, struct kmem_cache, list);
+ if (s->flags & SLAB_CHAINED)
+  return 0;

  for_each_online_node(node) {
   struct kmem_cache_node *n = get_node(s, node);
--
1.7.7.6
```