Subject: [PATCH v5 09/18] memcg: skip memcg kmem allocations in specified code regions
Posted by Glauber Costa on Fri, 19 Oct 2012 14:20:33 GMT
View Forum Message <> Reply to Message

This patch creates a mechanism that skip memcg allocations during
certain pieces of our core code. It basically works in the same way
as preempt_disable()/preempt_enable(): By marking a region under
which all allocations will be accounted to the root memcg.

We need this to prevent races in early cache creation, when we
allocate data using caches that are not necessarily created already.

[v2: wrap the whole enqueue process, INIT_WORK can alloc memory]

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/sched.h |  1 +
 mm/memcontrol.c       | 63 +++++++++++++++++++++++++++++++++++++++++++++++++++++--
 2 files changed, 62 insertions(+), 2 deletions(-)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index 0d907e1..9fad6c1 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1581,6 +1581,7 @@ struct task_struct {
   unsigned long nr_pages; /* uncharged usage */
   unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
  } memcg_batch;
+ unsigned int memcg_kmem_skip_account;
 #endif
 #ifdef CONFIG_HAVE_HW_BREAKPOINT
  atomic_t ptrace_bp_refcnt;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index ac2e621..3f46daa 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2950,6 +2950,41 @@ out:
  kfree(s->memcg_params);
 }
```

```
+/*
+ * During the creation a new cache, we need to disable our accounting mechanism
+ * altogether. This is true even if we are not creating, but rather just
+ * enqueing new caches to be created.
+ *
+ * This is because that process will trigger allocations; some visible, like
+ * explicit kmallocs to auxiliary data structures, name strings and internal
+ * cache structures; some well concealed, like INIT_WORK() that can allocate
+ * objects during debug.
+ *
+ * If any allocation happens during memcg_kmem_get_cache, we will recurse back
+ * to it. This may not be a bounded recursion: since the first cache creation
+ * failed to complete (waiting on the allocation), we'll just try to create the
+ * cache again, failing at the same point.
+ *
+ * memcg_kmem_get_cache is prepared to abort after seeing a positive count of
+ * memcg_kmem_skip_account. So we enclose anything that might allocate memory
+ * inside the following two functions.
+ */
+static void memcg_stop_kmem_account(void)
+{
+ if (!current->mm)
+  return;
+
+ current->memcg_kmem_skip_account++;
+}
+
+static void memcg_resume_kmem_account(void)
+{
+ if (!current->mm)
+  return;
+
+ current->memcg_kmem_skip_account--;
+}
+
 static char *memcg_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
 {
  char *name;
@@ -3009,7 +3044,10 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  if (new_cachep)
   goto out;

+ /* Don't block progress to enqueue caches for internal infrastructure */
+ memcg_stop_kmem_account();
  new_cachep = kmem_cache_dup(memcg, cachep);
+ memcg_resume_kmem_account();
```

```
  if (new_cachep == NULL) {
   new_cachep = cachep;
@@ -3047,8 +3085,8 @@ static void memcg_create_cache_work_func(struct work_struct *w)
  * Enqueue the creation of a per-memcg kmem_cache.
  * Called with rcu_read_lock.
  */
-static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
-           struct kmem_cache *cachep)
+static void __memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+      struct kmem_cache *cachep)
 {
  struct create_work *cw;

@@ -3067,6 +3105,24 @@ static void memcg_create_cache_enqueue(struct mem_cgroup
*memcg,
  schedule_work(&cw->work);
 }

+static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+           struct kmem_cache *cachep)
+{
+ /*
+  * We need to stop accounting when we kmalloc, because if the
+  * corresponding kmalloc cache is not yet created, the first allocation
+  * in __memcg_create_cache_enqueue will recurse.
+  *
+  * However, it is better to enclose the whole function. Depending on
+  * the debugging options enabled, INIT_WORK(), for instance, can
+  * trigger an allocation. This too, will make us recurse. Because at
+  * this point we can't allow ourselves back into memcg_kmem_get_cache,
+  * the safest choice is to do it like this, wrapping the whole function.
+  */
+ memcg_stop_kmem_account();
+ __memcg_create_cache_enqueue(memcg, cachep);
+ memcg_resume_kmem_account();
+}
 /*
  * Return the kmem_cache we're supposed to use for a slab allocation.
  * We try to use the current memcg's version of the cache.
@@ -3086,6 +3142,9 @@ struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache
*cachep,
  struct mem_cgroup *memcg;
  int idx;

+ if (!current->mm || current->memcg_kmem_skip_account)
+  return cachep;
+
  if (cachep->memcg_params && cachep->memcg_params->memcg)
```

return cachep;

--
1.7.11.7