Subject: [PATCH v5 2/2] decrement static keys on real destroy time
Posted by Glauber Costa on Fri, 11 May 2012 20:11:17 GMT
View Forum Message <> Reply to Message

We call the destroy function when a cgroup starts to be removed,
such as by a rmdir event.

However, because of our reference counters, some objects are still
inflight. Right now, we are decrementing the static_keys at destroy()
time, meaning that if we get rid of the last static_key reference,
some objects will still have charges, but the code to properly
uncharge them won't be run.

This becomes a problem specially if it is ever enabled again, because
now new charges will be added to the staled charges making keeping
it pretty much impossible.

We just need to be careful with the static branch activation:
since there is no particular preferred order of their activation,
we need to make sure that we only start using it after all
call sites are active. This is achieved by having a per-memcg
flag that is only updated after static_key_slow_inc() returns.
At this time, we are sure all sites are active.

This is made per-memcg, not global, for a reason:
it also has the effect of making socket accounting more
consistent. The first memcg to be limited will trigger static_key()
activation, therefore, accounting. But all the others will then be
accounted no matter what. After this patch, only limited memcgs
will have its sockets accounted.

[v2: changed a tcp limited flag for a generic proto limited flag]
[v3: update the current active flag only after the static_key update]
[v4: disarm_static_keys() inside free_work]
[v5: got rid of tcp_limit_mutex, now in the static_key interface]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Tejun Heo <tj@kernel.org>
CC: Li Zefan <lizefan@huawei.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Michal Hocko <mhocko@suse.cz>
---
 include/net/sock.h | 9 +++++++++
 mm/memcontrol.c | 26 ++++++++++++++++++++++++--
 net/ipv4/tcp_memcontrol.c | 32 ++++++++++++++++++++++++++++++-------
 3 files changed, 58 insertions(+), 9 deletions(-)

```
diff --git a/include/net/sock.h b/include/net/sock.h
index b3ebe6b..5c620bd 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -914,6 +914,15 @@ struct cg_proto {
  int   *memory_pressure;
  long   *sysctl_mem;
  /*
+  * active means it is currently active, and new sockets should
+  * be assigned to cgroups.
+  *
+  * activated means it was ever activated, and we need to
+  * disarm the static keys on destruction
+  */
+ bool   activated;
+ bool   active;
+ /*
    * memcg field is used to find which memcg we belong directly
    * Each memcg struct can hold more than one cg_proto, so container_of
    * won't really cut.
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 0b4b4c8..d1b0849 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -404,6 +404,7 @@ void sock_update_memcg(struct sock *sk)
 {
  if (mem_cgroup_sockets_enabled) {
   struct mem_cgroup *memcg;
+  struct cg_proto *cg_proto;

   BUG_ON(!sk->sk_prot->proto_cgroup);

@@ -423,9 +424,10 @@ void sock_update_memcg(struct sock *sk)

   rcu_read_lock();
   memcg = mem_cgroup_from_task(current);
-  if (!mem_cgroup_is_root(memcg)) {
+  cg_proto = sk->sk_prot->proto_cgroup(memcg);
+  if (!mem_cgroup_is_root(memcg) && cg_proto->active) {
    mem_cgroup_get(memcg);
-   sk->sk_cgrp = sk->sk_prot->proto_cgroup(memcg);
+   sk->sk_cgrp = cg_proto;
   }
   rcu_read_unlock();
  }
@@ -442,6 +444,14 @@ void sock_release_memcg(struct sock *sk)
  }
 }
```

```
+static void disarm_static_keys(struct mem_cgroup *memcg)
+{
+#ifdef CONFIG_INET
+ if (memcg->tcp_mem.cg_proto.activated)
+  static_key_slow_dec(&memcg_socket_limit_enabled);
+#endif
+}
+
 #ifdef CONFIG_INET
 struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 {
@@ -452,6 +462,11 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 }
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
+#else
+static inline void disarm_static_keys(struct mem_cgroup *memcg)
+{
+}
+
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4836,6 +4851,13 @@ static void free_work(struct work_struct *work)
  int size = sizeof(struct mem_cgroup);

  memcg = container_of(work, struct mem_cgroup, work_freeing);
+ /*
+  * We need to make sure that (at least for now), the jump label
+  * destruction code runs outside of the cgroup lock. schedule_work()
+  * will guarantee this happens. Be careful if you need to move this
+  * disarm_static_keys around
+  */
+ disarm_static_keys(memcg);
  if (size < PAGE_SIZE)
   kfree(memcg);
  else
diff --git a/net/ipv4/tcp_memcontrol.c b/net/ipv4/tcp_memcontrol.c
index 1517037..7ea4f79 100644
--- a/net/ipv4/tcp_memcontrol.c
+++ b/net/ipv4/tcp_memcontrol.c
@@ -74,9 +74,6 @@ void tcp_destroy_cgroup(struct mem_cgroup *memcg)
  percpu_counter_destroy(&tcp->tcp_sockets_allocated);

  val = res_counter_read_u64(&tcp->tcp_memory_allocated, RES_LIMIT);
-
- if (val != RESOURCE_MAX)
```

```
-  static_key_slow_dec(&memcg_socket_limit_enabled);
 }
 EXPORT_SYMBOL(tcp_destroy_cgroup);

@@ -107,10 +104,31 @@ static int tcp_update_limit(struct mem_cgroup *memcg, u64 val)
   tcp->tcp_prot_mem[i] = min_t(long, val >> PAGE_SHIFT,
          net->ipv4.sysctl_tcp_mem[i]);

- if (val == RESOURCE_MAX && old_lim != RESOURCE_MAX)
-  static_key_slow_dec(&memcg_socket_limit_enabled);
- else if (old_lim == RESOURCE_MAX && val != RESOURCE_MAX)
-  static_key_slow_inc(&memcg_socket_limit_enabled);
+ if (val == RESOURCE_MAX)
+  cg_proto->active = false;
+ else if (val != RESOURCE_MAX) {
+  /*
+   * ->activated needs to be written after the static_key update.
+   *  This is what guarantees that the socket activation function
+   *  is the last one to run. See sock_update_memcg() for details,
+   *  and note that we don't mark any socket as belonging to this
+   *  memcg until that flag is up.
+   *
+   *  We need to do this, because static_keys will span multiple
+   *  sites, but we can't control their order. If we mark a socket
+   *  as accounted, but the accounting functions are not patched in
+   *  yet, we'll lose accounting.
+   *
+   *  We never race with the readers in sock_update_memcg(), because
+   *  when this value change, the code to process it is not patched in
+   *  yet.
+   */
+  if (!cg_proto->activated) {
+   static_key_slow_inc(&memcg_socket_limit_enabled);
+   cg_proto->activated = true;
+  }
+  cg_proto->active = true;
+ }

  return 0;
 }
--
1.7.7.6
```