Subject: [PATCH v7 2/2] decrement static keys on real destroy time
Posted by Glauber Costa on Fri, 25 May 2012 09:32:08 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]
[v6: changed active and activated to a flags field, as suggested by akpm]
[v7: merged more comments from akpm]

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>
CC: Andrew Morton <akpm@linux-foundation.org>
---
 include/net/sock.h | 22 ++++++++++++++++++++
 mm/memcontrol.c | 31 ++++++++++++++++++++++++++++--

```
 net/ipv4/tcp_memcontrol.c |   34 ++++++++++++++++++++++++++++++++-------
 3 files changed, 78 insertions(+), 9 deletions(-)

diff --git a/include/net/sock.h b/include/net/sock.h
index b3ebe6b..d6a8ae3 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -46,6 +46,7 @@
 #include <linux/list_nulls.h>
 #include <linux/timer.h>
 #include <linux/cache.h>
+#include <linux/bitops.h>
 #include <linux/lockdep.h>
 #include <linux/netdevice.h>
 #include <linux/skbuff.h> /* struct sk_buff */
@@ -907,12 +908,23 @@ struct proto {
 #endif
 };

+/*
+ * Bits in struct cg_proto.flags
+ */
+enum cg_proto_flags {
+ /* Currently active and new sockets should be assigned to cgroups */
+ MEMCG_SOCK_ACTIVE,
+ /* It was ever activated; we must disarm static keys on destruction */
+ MEMCG_SOCK_ACTIVATED,
+};
+
 struct cg_proto {
   void   (*enter_memory_pressure)(struct sock *sk);
   struct res_counter *memory_allocated; /* Current allocated memory. */
   struct percpu_counter *sockets_allocated; /* Current number of sockets. */
   int   *memory_pressure;
   long   *sysctl_mem;
+ unsigned long  flags;
   /*
    * memcg field is used to find which memcg we belong directly
    * Each memcg struct can hold more than one cg_proto, so container_of
@@ -928,6 +940,16 @@ struct cg_proto {
 extern int proto_register(struct proto *prot, int alloc_slab);
 extern void proto_unregister(struct proto *prot);

+static inline bool memcg_proto_active(struct cg_proto *cg_proto)
+{
+ return test_bit(MEMCG_SOCK_ACTIVE, &cg_proto->flags);
+}
+
```

```
+static inline bool memcg_proto_activated(struct cg_proto *cg_proto)
+{
+ return test_bit(MEMCG_SOCK_ACTIVATED, &cg_proto->flags);
+}
+
 #ifdef SOCK_REFCNT_DEBUG
 static inline void sk_refcnt_debug_inc(struct sock *sk)
 {
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 0b4b4c8..788be2e 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) && memcg_proto_active(cg_proto)) {
    mem_cgroup_get(memcg);
-   sk->sk_cgrp = sk->sk_prot->proto_cgroup(memcg);
+   sk->sk_cgrp = cg_proto;
   }
   rcu_read_unlock();
  }
@@ -454,6 +456,19 @@ EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

+#if defined(CONFIG_INET) && defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
+static void disarm_sock_keys(struct mem_cgroup *memcg)
+{
+ if (!memcg_proto_activated(&memcg->tcp_mem.cg_proto))
+  return;
+ static_key_slow_dec(&memcg_socket_limit_enabled);
+}
+#else
+static void disarm_sock_keys(struct mem_cgroup *memcg)
+{
+}
```

```
+#endif
+
 static void drain_all_stock_async(struct mem_cgroup *memcg);

 static struct mem_cgroup_per_zone *
@@ -4836,6 +4851,18 @@ 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. This is because
+  * get_online_cpus(), which is called from the static_branch update,
+  * can't be called inside the cgroup_lock. cpusets are the ones
+  * enforcing this dependency, so if they ever change, we might as well.
+  *
+  * schedule_work() will guarantee this happens. Be careful if you need
+  * to move this code around, and make sure it is outside
+  * the cgroup_lock.
+  */
+ disarm_sock_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..b6f3583 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 +104,33 @@ 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)
+  clear_bit(MEMCG_SOCK_ACTIVE, &cg_proto->flags);
```

```
+ else if (val != RESOURCE_MAX) {
+  /*
+   * The active bit 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.
+   *
+   * The activated bit is used to guarantee that no two writers
+   * will do the update in the same memcg. Without that, we can't
+   * properly shutdown the static key.
+   */
+  if (!test_and_set_bit(MEMCG_SOCK_ACTIVATED, &cg_proto->flags))
+   static_key_slow_inc(&memcg_socket_limit_enabled);
+  set_bit(MEMCG_SOCK_ACTIVE, &cg_proto->flags);
+ }

  return 0;
 }
--
1.7.7.6
```