Subject: [PATCH v5 8/8] Disable task moving when using kernel memory accounting
Posted by Glauber Costa on Tue, 04 Oct 2011 12:18:00 GMT
View Forum Message <> Reply to Message

Since this code is still experimental, we are leaving the exact
details of how to move tasks between cgroups when kernel memory
accounting is used as future work.

For now, we simply disallow movement if there are any pending
accounted memory.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Hiroyouki Kamezawa <kamezawa.hiroyu@jp.fujitsu.com>
---
 include/net/tcp.h | 1 +
 mm/memcontrol.c | 37 ++++++++++++++++++++++++++----------
 2 files changed, 23 insertions(+), 15 deletions(-)

```
diff --git a/include/net/tcp.h b/include/net/tcp.h
index 716a42e..62ad282 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -257,6 +257,7 @@ struct mem_cgroup;
 struct tcp_memcontrol {
  /* per-cgroup tcp memory pressure knobs */
  int tcp_max_memory;
+ atomic_t refcnt;
  atomic_long_t tcp_memory_allocated;
  struct percpu_counter tcp_sockets_allocated;
  /* those two are read-mostly, leave them at the end */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index f178a64..db1f511 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -328,28 +328,17 @@ void sock_update_memcg(struct sock *sk)

  rcu_read_lock();
  sk->sk_cgrp = mem_cgroup_from_task(current);
-
- /*
-  * We don't need to protect against anything task-related, because
-  * we are basically stuck with the sock pointer that won't change,
-  * even if the task that originated the socket changes cgroups.
-  *
-  * What we do have to guarantee, is that the chain leading us to
-  * the top level won't change under our noses. Incrementing the
-  * reference count via cgroup_exclude_rmdir guarantees that.
```

```
-  */
- cgroup_exclude_rmdir(mem_cgroup_css(sk->sk_cgrp));
  rcu_read_unlock();
 }

 void sock_release_memcg(struct sock *sk)
 {
- cgroup_release_and_wakeup_rmdir(mem_cgroup_css(sk->sk_cgrp));
 }

 void memcg_sock_mem_alloc(struct mem_cgroup *memcg, struct proto *prot,
      int amt, int *parent_failure)
 {
+ atomic_inc(&memcg->tcp.refcnt);
  memcg = parent_mem_cgroup(memcg);
  for (; memcg != NULL; memcg = parent_mem_cgroup(memcg)) {
   long alloc;
@@ -368,9 +357,12 @@ EXPORT_SYMBOL(memcg_sock_mem_alloc);

 void memcg_sock_mem_free(struct mem_cgroup *memcg, struct proto *prot, int amt)
 {
- memcg = parent_mem_cgroup(memcg);
- for (; memcg != NULL; memcg = parent_mem_cgroup(memcg))
-  atomic_long_sub(amt, prot->memory_allocated(memcg));
+ struct mem_cgroup *parent;
+ parent = parent_mem_cgroup(memcg);
+ for (; parent != NULL; parent = parent_mem_cgroup(parent))
+  atomic_long_sub(amt, prot->memory_allocated(parent));
+
+ atomic_dec(&memcg->tcp.refcnt);
 }
 EXPORT_SYMBOL(memcg_sock_mem_free);

@@ -488,6 +480,7 @@ static void tcp_create_cgroup(struct mem_cgroup *cg, struct
cgroup_subsys *ss)
 {
  cg->tcp.tcp_memory_pressure = 0;
  atomic_long_set(&cg->tcp.tcp_memory_allocated, 0);
+ atomic_set(&cg->tcp.refcnt, 0);
  percpu_counter_init(&cg->tcp.tcp_sockets_allocated, 0);
 }

@@ -5633,6 +5626,12 @@ static int mem_cgroup_can_attach(struct cgroup_subsys *ss,
  int ret = 0;
  struct mem_cgroup *mem = mem_cgroup_from_cont(cgroup);

+ if (atomic_read(&mem->tcp.refcnt)) {
+  printk(KERN_WARNING "Can't move tasks between cgroups: "
```

```
+   "Kernel memory held. task: %s\n", p->comm);
+  return 1;
+ }
+
  if (mem->move_charge_at_immigrate) {
   struct mm_struct *mm;
   struct mem_cgroup *from = mem_cgroup_from_task(p);
@@ -5803,6 +5802,14 @@ static int mem_cgroup_can_attach(struct cgroup_subsys *ss,
     struct cgroup *cgroup,
     struct task_struct *p)
 {
+ struct mem_cgroup *mem = mem_cgroup_from_cont(cgroup);
+
+ if (atomic_read(&mem->tcp.refcnt)) {
+  printk(KERN_WARNING "Can't move tasks between cgroups: "
+   "Kernel memory held. task: %s\n", p->comm);
+  return 1;
+ }
+
  return 0;
 }
 static void mem_cgroup_cancel_attach(struct cgroup_subsys *ss,
--
1.7.6
```