
Subject: [RFC] memcg: propagate kmem limiting information to children
Posted by [Glauber Costa](#) on Mon, 21 May 2012 11:34:50 GMT
[View Forum Message](#) <> [Reply to Message](#)

Hi,

So far, my code passed the basic hierarchical tests. However, it failed to present satisfactory behavior in the following scenario:

-> /cgroups/memory/A/B/C

- * kmem limit set at A
- * A and B empty taskwise
- * bash in C does find /

Because kmem_accounted is a boolean that was not set for C, no accounting would be done. This is, however, not what we expect.

The basic idea, is that when a cgroup is limited, we walk the tree upwards (something Kame and I already thought about doing for other purposes), and make sure that we store the information about the parent being limited in kmem_accounted (that is turned into a bitmap: two booleans would not be space efficient). The code for that is taken from sched/core.c. My reasons for not putting it into a common place is to dodge the type issues that would arise from a common implementation between memcg and the scheduler - but I think that it should ultimately happen, so if you want me to do it now, let me know.

We do the reverse operation when a formerly limited cgroup becomes unlimited.

I am sending this as an early preview of this solution. I would like to know what you think.

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>

mm/memcontrol.c | 147 +++-----
1 files changed, 130 insertions(+), 17 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index a06eb3f..47d5734 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c

```

@@ -259,6 +259,9 @@ struct mem_cgroup {
    * the counter to account for kernel memory usage.
    */
    struct res_counter kmem;
+
+ struct list_head children;
+ struct list_head siblings;
    /*
    * Per cgroup active and inactive list, similar to the
    * per zone LRU lists.
@@ -274,7 +277,11 @@ struct mem_cgroup {
    * Should the accounting and control be hierarchical, per subtree?
    */
    bool use_hierarchy;
- bool kmem_accounted;
+ /*
+  * bit0: accounted by this cgroup
+  * bit1: accounted by a parent.
+  */
+ volatile unsigned long kmem_accounted;

    bool oom_lock;
    atomic_t under_oom;
@@ -332,6 +339,9 @@ struct mem_cgroup {
#endif
};

+#define KMEM_ACCOUNTED_THIS 0
+#define KMEM_ACCOUNTED_PARENT 1
+
int memcg_css_id(struct mem_cgroup *memcg)
{
    return css_id(&memcg->css);
@@ -474,7 +484,7 @@ void sock_release_memcg(struct sock *sk)

static void disarm_static_keys(struct mem_cgroup *memcg)
{
- if (memcg->kmem_accounted)
+ if (memcg->kmem_accounted & (1 << KMEM_ACCOUNTED_THIS))
    static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
    /*
    * This check can't live in kmem destruction function,
@@ -4472,6 +4482,109 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
    len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
    return simple_read_from_buffer(buf, nbytes, ppos, str, len);
}
+

```

```

+typedef int (*memcg_visitor)(struct mem_cgroup*, void *);
+
+/*
+ * This is mostly "inspired" by the code in sched/core.c. I decided to copy it,
+ * instead of factoring it, because of all the typing issues we'd run into.
+ * In particular, grabbing the parent is very different for memcg, because we
+ * may or may not have hierarchy, while cpu cgroups always do. That would lead
+ * to either indirect calls - this is not a fast path for us, but can be for
+ * the scheduler - or a big and ugly macro.
+ *
+ * If we ever get rid of hierarchy, we could iterate over struct cgroup, and
+ * then it would cease to be a problem.
+ */
+int walk_tree_from(struct mem_cgroup *from,
+    memcg_visitor down, memcg_visitor up, void *data)
+{
+    struct mem_cgroup *parent, *child;
+    int ret;
+
+
+    parent = from;
+down:
+    ret = (*down)(parent, data);
+    if (ret)
+        goto out;
+
+    list_for_each_entry_rcu(child, &parent->children, siblings) {
+        parent = child;
+        goto down;
+    }
+up:
+    continue;
+}
+    ret = (*up)(parent, data);
+    if (ret || parent == from)
+        goto out;
+
+    child = parent;
+    parent = parent_mem_cgroup(parent);
+    if (parent)
+        goto up;
+out:
+    return ret;
+}
+
+static int memcg_nop(struct mem_cgroup *memcg, void *data)
+{
+    return 0;
+}

```

```

+}
+
+static int memcg_parent_account(struct mem_cgroup *memcg, void *data)
+{
+ if (memcg == data)
+ return 0;
+
+ set_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+ return 0;
+}
+
+static int memcg_parent_no_account(struct mem_cgroup *memcg, void *data)
+{
+ if (memcg == data)
+ return 0;
+
+ clear_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+ /*
+  * Stop propagation if we are accounted: our children should
+  * be parent-accounted
+  */
+ return memcg->kmem_accounted & (1 << KMEM_ACCOUNTED_THIS);
+}
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static void mem_cgroup_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
+{
+ mutex_lock(&set_limit_mutex);
+ if (!test_and_set_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted) &&
+ val != RESOURCE_MAX) {
+
+ /*
+  * Once enabled, can't be disabled. We could in theory
+  * disable it if we haven't yet created any caches, or
+  * if we can shrink them all to death.
+  *
+  * But it is not worth the trouble
+  */
+ static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
+
+ rcu_read_lock();
+ walk_tree_from(memcg, memcg_parent_account, memcg_nop, memcg);
+ rcu_read_unlock();
+ } else if (test_and_clear_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted)
+ && val == RESOURCE_MAX) {
+
+ rcu_read_lock();
+ walk_tree_from(memcg, memcg_parent_no_account, memcg_nop, memcg);

```



```

int cpu;
@@ -5645,6 +5749,10 @@ mem_cgroup_create(struct cgroup *cont)
    * mem_cgroup(see mem_cgroup_put).
    */
    mem_cgroup_get(parent);
+
+ mutex_lock(&memcg_list_mutex);
+ list_add_rcu(&memcg->siblings, &parent->children);
+ mutex_unlock(&memcg_list_mutex);
    } else {
        res_counter_init(&memcg->res, NULL);
        res_counter_init(&memcg->memsw, NULL);
@@ -5656,7 +5764,6 @@ mem_cgroup_create(struct cgroup *cont)
    if (parent)
        memcg->swappiness = mem_cgroup_swappiness(parent);
    atomic_set(&memcg->refcnt, 1);
- memcg->kmem_accounted = false;
    memcg->move_charge_at_immigrate = 0;
    mutex_init(&memcg->thresholds_lock);
    spin_lock_init(&memcg->move_lock);
@@ -5688,9 +5795,15 @@ static int mem_cgroup_pre_destroy(struct cgroup *cont)
static void mem_cgroup_destroy(struct cgroup *cont)
{
    struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);

    kmem_cgroup_destroy(memcg);

+ mutex_lock(&memcg_list_mutex);
+ if (parent)
+ list_del_rcu(&memcg->siblings);
+ mutex_unlock(&memcg_list_mutex);
+
    mem_cgroup_put(memcg);
}

--
1.7.7.6

```
