

---

Subject: [PATCH v4 04/14] kmem accounting basic infrastructure  
Posted by [Glauber Costa](#) on Mon, 08 Oct 2012 10:06:10 GMT  
[View Forum Message](#) <> [Reply to Message](#)

---

This patch adds the basic infrastructure for the accounting of the slab caches. To control that, the following files are created:

- \* memory.kmem.usage\_in\_bytes
- \* memory.kmem.limit\_in\_bytes
- \* memory.kmem.failcnt
- \* memory.kmem.max\_usage\_in\_bytes

They have the same meaning of their user memory counterparts. They reflect the state of the "kmem" res\_counter.

Per cgroup slab memory accounting is not enabled until a limit is set for the group. Once the limit is set the accounting cannot be disabled for that group. This means that after the patch is applied, no behavioral changes exists for whoever is still using memcg to control their memory usage, until memory.kmem.limit\_in\_bytes is set for the first time.

We always account to both user and kernel resource\_counters. This effectively means that an independent kernel limit is in place when the limit is set to a lower value than the user memory. A equal or higher value means that the user limit will always hit first, meaning that kmem is effectively unlimited.

People who want to track kernel memory but not limit it, can set this limit to a very high number (like RESOURCE\_MAX - 1page - that no one will ever hit, or equal to the user memory)

[ v4: make kmem files part of the main array;  
do not allow limit to be set for non-empty cgroups ]

Signed-off-by: Glauber Costa <glommer@parallels.com>  
CC: Michal Hocko <mhocko@suse.cz>  
CC: Johannes Weiner <hannes@cmpxchg.org>  
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

---  
mm/memcontrol.c | 123 ++++++-----  
1 file changed, 122 insertions(+), 1 deletion(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 71d259e..ba855cc 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -266,6 +266,10 @@ struct mem_cgroup {
```

```

};

/*
+ * the counter to account for kernel memory usage.
+ */
+ struct res_counter kmem;
+ /*
+ * Per cgroup active and inactive list, similar to the
+ * per zone LRU lists.
+ */
@@ -280,6 +284,7 @@ struct mem_cgroup {
+ * Should the accounting and control be hierarchical, per subtree?
+ */
+ bool use_hierarchy;
+ unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */

+ bool oom_lock;
+ atomic_t under_oom;
@@ -332,6 +337,26 @@ struct mem_cgroup {
+ #endif
+ };

+ /* internal only representation about the status of kmem accounting. */
+ enum {
+ KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
+ };
+
+ /* first bit */
+ #define KMEM_ACCOUNTED_MASK 0x1
+
+ #ifdef CONFIG_MEMCG_KMEM
+ static void memcg_kmem_set_active(struct mem_cgroup *memcg)
+ {
+ set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+ }
+
+ static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
+ {
+ return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+ }
+ #endif
+
+ /* Stuffs for move charges at task migration. */
+ /*
+ * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -390,6 +415,7 @@ enum res_type {
+ _MEM,
+ _MEMSWAP,

```

```

    _OOM_TYPE,
+ _KMEM,
};

#define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
@@ -1433,6 +1459,10 @@ done:
    res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
    res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
    res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
+ printk(KERN_INFO "kmem: usage %lluKB, limit %lluKB, failcnt %llu\n",
+ res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
+ res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
+ res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
}

/*
@@ -3940,6 +3970,9 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype *cft,
    else
        val = res_counter_read_u64(&memcg->memsw, name);
    break;
+ case _KMEM:
+ val = res_counter_read_u64(&memcg->kmem, name);
+ break;
    default:
        BUG();
}
@@ -3947,6 +3980,58 @@ 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);
}
+
+static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
+{
+ int ret = -EINVAL;
+ #ifdef CONFIG_MEMCG_KMEM
+ struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ /*
+  * For simplicity, we won't allow this to be disabled. It also can't
+  * be changed if the cgroup has children already, or if tasks had
+  * already joined.
+  *
+  * If tasks join before we set the limit, a person looking at
+  * kmem.usage_in_bytes will have no way to determine when it took
+  * place, which makes the value quite meaningless.
+  *
+  * After it first became limited, changes in the value of the limit are
+  * of course permitted.

```

```

+ *
+ * Taking the cgroup_lock is really offensive, but it is so far the only
+ * way to guarantee that no children will appear. There are plenty of
+ * other offenders, and they should all go away. Fine grained locking
+ * is probably the way to go here. When we are fully hierarchical, we
+ * can also get rid of the use_hierarchy check.
+ */
+ cgroup_lock();
+ mutex_lock(&set_limit_mutex);
+ if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
+   if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
+     !list_empty(&cont->children))) {
+     ret = -EBUSY;
+     goto out;
+   }
+   ret = res_counter_set_limit(&memcg->kmem, val);
+   if (ret)
+     goto out;
+ }
+ memcg_kmem_set_active(memcg);
+ } else
+   ret = res_counter_set_limit(&memcg->kmem, val);
+out:
+ mutex_unlock(&set_limit_mutex);
+ cgroup_unlock();
+ #endif
+ return ret;
+}
+
+static void memcg_propagate_kmem(struct mem_cgroup *memcg,
+ struct mem_cgroup *parent)
+{
+ memcg->kmem_accounted = parent->kmem_accounted;
+}
+
+/*
+ * The user of this function is...
+ * RES_LIMIT.
@@ -3978,8 +4063,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    break;
    if (type == _MEM)
        ret = mem_cgroup_resize_limit(memcg, val);
- else
+ else if (type == _MEMSWAP)
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
+ else if (type == _KMEM)
+   ret = memcg_update_kmem_limit(cont, val);
+ else

```

```

+ return -EINVAL;
  break;
  case RES_SOFT_LIMIT:
    ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4045,12 +4134,16 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
  case RES_MAX_USAGE:
    if (type == _MEM)
      res_counter_reset_max(&memcg->res);
+ else if (type == _KMEM)
+ res_counter_reset_max(&memcg->kmem);
  else
    res_counter_reset_max(&memcg->memsw);
  break;
  case RES_FAILCNT:
    if (type == _MEM)
      res_counter_reset_failcnt(&memcg->res);
+ else if (type == _KMEM)
+ res_counter_reset_failcnt(&memcg->kmem);
  else
    res_counter_reset_failcnt(&memcg->memsw);
  break;
@@ -4728,6 +4821,31 @@ static struct cftype mem_cgroup_files[] = {
  .read = mem_cgroup_read,
},
#endif
#ifdef CONFIG_MEMCG_KMEM
+ {
+ .name = "kmem.limit_in_bytes",
+ .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+ .write_string = mem_cgroup_write,
+ .read = mem_cgroup_read,
+ },
+ {
+ .name = "kmem.usage_in_bytes",
+ .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
+ .read = mem_cgroup_read,
+ },
+ {
+ .name = "kmem.failcnt",
+ .private = MEMFILE_PRIVATE(_KMEM, RES_FAILCNT),
+ .trigger = mem_cgroup_reset,
+ .read = mem_cgroup_read,
+ },
+ {
+ .name = "kmem.max_usage_in_bytes",
+ .private = MEMFILE_PRIVATE(_KMEM, RES_MAX_USAGE),
+ .trigger = mem_cgroup_reset,

```

```

+ .read = mem_cgroup_read,
+ },
+#endif
  { }, /* terminate */
};

@@ -4973,6 +5091,7 @@ mem_cgroup_create(struct cgroup *cont)
  if (parent && parent->use_hierarchy) {
    res_counter_init(&memcg->res, &parent->res);
    res_counter_init(&memcg->memsw, &parent->memsw);
+ res_counter_init(&memcg->kmem, &parent->kmem);
  /*
   * We increment refcnt of the parent to ensure that we can
   * safely access it on res_counter_charge/uncharge.
@@ -4980,9 +5099,11 @@ mem_cgroup_create(struct cgroup *cont)
  * mem_cgroup(see mem_cgroup_put).
  */
  mem_cgroup_get(parent);
+ memcg_propagate_kmem(memcg, parent);
  } else {
    res_counter_init(&memcg->res, NULL);
    res_counter_init(&memcg->memsw, NULL);
+ res_counter_init(&memcg->kmem, NULL);
  /*
   * Deeper hierachy with use_hierarchy == false doesn't make
   * much sense so let cgroup subsystem know about this

```

--

1.7.11.4