
Subject: [PATCH v3 19/28] slab: per-memcg accounting of slab caches

Posted by [Glauber Costa](#) on Fri, 25 May 2012 13:03:39 GMT

[View Forum Message](#) <> [Reply to Message](#)

This patch changes allocation of a slab object to a particular memcg.

The cache is selected with `mem_cgroup_get_kmem_cache()`, which is the biggest overhead we pay here, because it happens at all allocations. However, other than forcing a function call, this function is not very expensive, and try to return as soon as we realize we are not a memcg cache.

The charge/uncharge functions are heavier, but are only called for new page allocations.

Code is heavily inspired by Suleiman's, with adaptations to the patchset and minor simplifications by me.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Signed-off-by: Suleiman Souhlal <suleiman@google.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>

```
include/linux/slab_def.h |  62 ++++++-----  
mm/slab.c           |  98 ++++++-----  
2 files changed, 151 insertions(+), 9 deletions(-)
```

```
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h  
index 7c0cdd6..ea9054a 100644  
--- a/include/linux/slab_def.h  
+++ b/include/linux/slab_def.h  
@@ -225,4 +225,66 @@ found:  
  
#endif /* CONFIG_NUMA */  
  
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM  
+  
+void kmem_cache_drop_ref(struct kmem_cache *cachep);  
+  
+static inline void  
+kmem_cache_get_ref(struct kmem_cache *cachep)  
+{  
+ if (cachep->memcg_params.id == -1 &&  
+     unlikely(!atomic_add_unless(&cachep->memcg_params.refcnt, 1, 0)))
```

```

+ BUG();
+}
+
+static inline void
+mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
+{
+ rCU_read_unlock();
+}
+
+static inline void
+mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
+{
+ /*
+ * Make sure the cache doesn't get freed while we have interrupts
+ * enabled.
+ */
+ kmem_cache_get_ref(cachep);
+}
+
+static inline void
+mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
+{
+ kmem_cache_drop_ref(cachep);
+}
+
+/*#else /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
+static inline void
+kmem_cache_get_ref(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+kmem_cache_drop_ref(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+mem_cgroup_put_kmem_cache(struct kmem_cache *cachep)
+{
+}
+
+static inline void
+mem_cgroup_kmem_cache_prepare_sleep(struct kmem_cache *cachep)
+{
+}
+
+static inline void

```

```

+mem_cgroup_kmem_cache_finish_sleep(struct kmem_cache *cachep)
+{
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
#endif /* _LINUX_SLAB_DEF_H */
diff --git a/mm/slab.c b/mm/slab.c
index e2227de..16ad229 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1845,20 +1845,28 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t
flags, int nodeid)
if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
flags |= __GFP_RECLAIMABLE;

+ nr_pages = (1 << cachep->gfporder);
+ if (!mem_cgroup_charge_slab(cachep, flags, nr_pages * PAGE_SIZE))
+ return NULL;
+
page = alloc_pages_exact_node(nodeid, flags | __GFP_NOTRACK, cachep->gfporder);
if (!page) {
if (!(flags & __GFP_NOWARN) && printk_ratelimit())
slab_out_of_memory(cachep, flags, nodeid);
+
+ mem_cgroup_uncharge_slab(cachep, nr_pages * PAGE_SIZE);
return NULL;
}

- nr_pages = (1 << cachep->gfporder);
if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
add_zone_page_state(page_zone(page),
NR_SLAB_RECLAIMABLE, nr_pages);
else
add_zone_page_state(page_zone(page),
NR_SLAB_UNRECLAIMABLE, nr_pages);
+
+ kmem_cache_get_ref(cachep);
+
for (i = 0; i < nr_pages; i++)
__SetPageSlab(page + i);

@@ -1874,6 +1882,14 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t
flags, int nodeid)
return page_address(page);
}

+ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+void kmem_cache_drop_ref(struct kmem_cache *cachep)

```

```

+{
+ if (cachep->memcg_params.id == -1)
+ atomic_dec(&cachep->memcg_params.refcnt);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
/*
 * Interface to system's page release.
 */
@@ -1891,6 +1907,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
else
    sub_zone_page_state(page_zone(page),
        NR_SLAB_UNRECLAIMABLE, nr_freed);
+ mem_cgroup_uncharge_slab(cachep, i * PAGE_SIZE);
+ kmem_cache_drop_ref(cachep);
while (i--) {
    BUG_ON(!PageSlab(page));
    __ClearPageSlab(page);
@@ -2855,7 +2873,6 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
if (cachep->memcg_params.id == -1)
    kfree(cachep->name);
#endif
-
    __kmem_cache_destroy(cachep);
    mutex_unlock(&cache_chain_mutex);
    put_online_cpus();
@@ -3061,8 +3078,10 @@ static int cache_grow(struct kmem_cache *cachep,
offset *= cachep->colour_off;

- if (local_flags & __GFP_WAIT)
+ if (local_flags & __GFP_WAIT) {
    local_irq_enable();
+ mem_cgroup_kmem_cache_prepare_sleep(cachep);
+ }

/*
 * The test for missing atomic flag is performed here, rather than
@@ -3091,8 +3110,10 @@ static int cache_grow(struct kmem_cache *cachep,
cache_init_objs(cachep, slabp);

- if (local_flags & __GFP_WAIT)
+ if (local_flags & __GFP_WAIT) {
    local_irq_disable();
+ mem_cgroup_kmem_cache_finish_sleep(cachep);
+ }
    check_irq_off();

```

```

spin_lock(&l3->list_lock);

@@ -3105,8 +3126,10 @@ static int cache_grow(struct kmem_cache *cachep,
opps1:
    kmem_freepages(cachep, objp);
failed:
- if (local_flags & __GFP_WAIT)
+ if (local_flags & __GFP_WAIT) {
    local_irq_disable();
+ mem_cgroup_kmem_cache_finish_sleep(cachep);
+ }
    return 0;
}

@@ -3867,11 +3890,15 @@ static inline void __cache_free(struct kmem_cache *cachep, void
*objp,
 */
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
{
- void *ret = __cache_alloc(cachep, flags, __builtin_return_address(0));
+ void *ret;
+
+ rCU_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rCU_read_unlock();
+ ret = __cache_alloc(cachep, flags, __builtin_return_address(0));

trace_kmem_cache_alloc(_RET_IP_, ret,
    obj_size(cachep), cachep->buffer_size, flags);
-
    return ret;
}
EXPORT_SYMBOL(kmem_cache_alloc);
@@ -3882,6 +3909,10 @@ kmem_cache_alloc_trace(size_t size, struct kmem_cache *cachep,
gfp_t flags)
{
    void *ret;

+ rCU_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rCU_read_unlock();
+
    ret = __cache_alloc(cachep, flags, __builtin_return_address(0));

trace_kmalloc(_RET_IP_, ret,
@@ -3894,13 +3925,17 @@ EXPORT_SYMBOL(kmem_cache_alloc_trace);
#ifndef CONFIG_NUMA
void *kmem_cache_alloc_node(struct kmem_cache *cachep, gfp_t flags, int nodeid)

```

```

{
- void *ret = __cache_alloc_node(cachep, flags, nodeid,
+ void *ret;
+
+ rCU_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rCU_read_unlock();
+ ret = __cache_alloc_node(cachep, flags, nodeid,
    __builtin_return_address(0));

trace_kmem_cache_alloc_node(_RET_IP_, ret,
    obj_size(cachep), cachep->buffer_size,
    flags, nodeid);

-
return ret;
}
EXPORT_SYMBOL(kmem_cache_alloc_node);
@@ -3913,6 +3948,9 @@ void *kmem_cache_alloc_node_trace(size_t size,
{
void *ret;

+ rCU_read_lock();
+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+ rCU_read_unlock();
ret = __cache_alloc_node(cachep, flags, nodeid,
    __builtin_return_address(0));
trace_kmalloc_node(_RET_IP_, ret,
@@ -4021,9 +4059,33 @@ void kmem_cache_free(struct kmem_cache *cachep, void *objp)

local_irq_save(flags);
debug_check_no_locks_freed(objp, obj_size(cachep));
+
+ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+{
+ struct kmem_cache *actual_cachep;
+
+ actual_cachep = virt_to_cache(objp);
+ if (actual_cachep != cachep) {
+ VM_BUG_ON(actual_cachep->memcg_params.id != -1);
+ cachep = actual_cachep;
+ }
+ /*
+ * Grab a reference so that the cache is guaranteed to stay
+ * around.
+ * If we are freeing the last object of a dead memcg cache,
+ * the kmem_cache_drop_ref() at the end of this function
+ * will end up freeing the cache.
+ */

```

```

+ kmem_cache_get_ref(cachep);
+ }
+endif
+
if (!(cachep->flags & SLAB_DEBUG_OBJECTS))
    debug_check_no_obj_freed(objp, obj_size(cachep));
    __cache_free(cachep, objp, __builtin_return_address(0));
+
+ kmem_cache_drop_ref(cachep);
+
local_irq_restore(flags);

trace_kmem_cache_free(_RET_IP_, objp);
@@ -4051,9 +4113,19 @@ void kfree(const void *objp)
local_irq_save(flags);
kfree_debugcheck(objp);
c = virt_to_cache(objp);
+
+ /*
+ * Grab a reference so that the cache is guaranteed to stay around.
+ * If we are freeing the last object of a dead memcg cache, the
+ * kmem_cache_drop_ref() at the end of this function will end up
+ * freeing the cache.
+ */
+ kmem_cache_get_ref(c);
+
debug_check_no_locks_freed(objp, obj_size(c));
debug_check_no_obj_freed(objp, obj_size(c));
__cache_free(c, (void *)objp, __builtin_return_address(0));
+ kmem_cache_drop_ref(c);
local_irq_restore(flags);
}
EXPORT_SYMBOL(kfree);
@@ -4322,6 +4394,13 @@ static void cache_reap(struct work_struct *w)
list_for_each_entry(searchp, &cache_chain, next) {
    check_irq_on();

+ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* For memcg caches, make sure we only reap the active ones. */
+ if (searchp->memcg_params.id == -1 &&
+     !atomic_add_unless(&searchp->memcg_params.refcnt, 1, 0))
+     continue;
+endif
+
/*
 * We only take the I3 lock if absolutely necessary and we
 * have established with reasonable certainty that
@@ -4354,6 +4433,7 @@ static void cache_reap(struct work_struct *w)

```

```
    STATS_ADD_REAPED(searchp, freed);
}
next:
+ kmem_cache_drop_ref(searchp);
    cond_resched();
}
check_irq_on();
--
```

1.7.7.6
