Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Glauber Costa on Mon, 24 Sep 2012 08:46:35 GMT
View Forum Message <> Reply to Message

On 09/21/2012 10:32 PM, Tejun Heo wrote:

```
> On Tue, Sep 18, 2012 at 06:12:00PM +0400, Glauber Costa wrote:
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 04851bb..1cce5c3 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -339,6 +339,11 @@ struct mem_cgroup {
>>  #ifdef CONFIG_INET
>>   struct tcp_memcontrol tcp_mem;
>>  #endif
>> +
>> +#ifdef CONFIG_MEMCG_KMEM
>> + /* Slab accounting */
>> + struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
>> +#endif
>
> Bah, 400 entry array in struct mem_cgroup.  Can't we do something a
> bit more flexible?
>
```

I guess. I still would like it to be an array, so we can easily access
its fields. There are two ways around this:

1) Do like the events mechanism and allocate this in a separate
structure. Add a pointer chase in the access, and I don't think it helps
much because it gets allocated anyway. But we could at least
defer it to the time when we limit the cache.

2) The indexes are only assigned after the slab is FULL. At that time, a
lot of the caches are already initialized. We can, for instance, allow
for "twice the number we have in the system", which already provides
room for a couple of more appearing. Combining with the 1st approach, we
can defer it to limit-time, and then allow for, say, 50 % more caches
than we already have. The pointer chasing may very well be worth it...

```
>> +static char *memcg_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
>> +{
>> + char *name;
>> + struct dentry *dentry;
>> +
>> + rcu_read_lock();
>> + dentry = rcu_dereference(memcg->css.cgroup->dentry);
>> + rcu_read_unlock();
>> +
>> + BUG_ON(dentry == NULL);
>> +
>> + name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
>> +     cachep->name, css_id(&memcg->css), dentry->d_name.name);
>
> Maybe including full path is better, I don't know.
```

It can get way too big.

```
>> +static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
>> +           struct kmem_cache *cachep)
>> +{
>> + struct create_work *cw;
>> + unsigned long flags;
>> +
>> + spin_lock_irqsave(&cache_queue_lock, flags);
>> + list_for_each_entry(cw, &create_queue, list) {
>> +  if (cw->memcg == memcg && cw->cachep == cachep) {
>> +   spin_unlock_irqrestore(&cache_queue_lock, flags);
>> +   return;
>> +  }
>> + }
>> + spin_unlock_irqrestore(&cache_queue_lock, flags);
>> +
>> + /* The corresponding put will be done in the workqueue. */
>> + if (!css_tryget(&memcg->css))
>> +  return;
>> +
>> + cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
>> + if (cw == NULL) {
>> +  css_put(&memcg->css);
>> +  return;
>> + }
>> +
>> + cw->memcg = memcg;
>> + cw->cachep = cachep;
>> + spin_lock_irqsave(&cache_queue_lock, flags);
>> + list_add_tail(&cw->list, &create_queue);
>> + spin_unlock_irqrestore(&cache_queue_lock, flags);
>> +
>> + schedule_work(&memcg_create_cache_work);
>> +}
>
> Why create your own worklist and flush mechanism?  Just embed a work
> item in create_work and use a dedicated workqueue for flushing.
```

I'll take a look at this.

```
>
>> +/*
>> + * Return the kmem_cache we're supposed to use for a slab allocation.
>> + * We try to use the current memcg's version of the cache.
>> + *
>> + * If the cache does not exist yet, if we are the first user of it,
>> + * we either create it immediately, if possible, or create it asynchronously
>> + * in a workqueue.
>> + * In the latter case, we will let the current allocation go through with
>> + * the original cache.
>> + *
>> + * Can't be called in interrupt context or from kernel threads.
>> + * This function needs to be called with rcu_read_lock() held.
>> + */
>> +struct kmem_cache *__memcg_kmem_get_cache(struct kmem_cache *cachep,
>> +       gfp_t gfp)
>> +{
>> + struct mem_cgroup *memcg;
>> + int idx;
>> + struct task_struct *p;
>> +
>> + if (cachep->memcg_params.memcg)
>> +  return cachep;
>> +
>> + idx = cachep->memcg_params.id;
>> + VM_BUG_ON(idx == -1);
>> +
>> + rcu_read_lock();
>> + p = rcu_dereference(current->mm->owner);
>> + memcg = mem_cgroup_from_task(p);
>> + rcu_read_unlock();
>> +
>> + if (!memcg_can_account_kmem(memcg))
>> +  return cachep;
>> +
>> + if (memcg->slabs[idx] == NULL) {
>> +  memcg_create_cache_enqueue(memcg, cachep);
>
> Do we want to wait for the work item if @gfp allows?
>
```

I tried this once, and it got complicated enough that I deemed as "not worth it". I honestly don't remember much of the details now, it was one of the first things I tried, and a bunch of time has passed. If you think it is absolutely worth it, I can try it again. But at the very best, I view this as an optimization.