Subject: Re: [PATCH v3 06/16] memcg: infrastructure to match an allocation to the right cache
Posted by Tejun Heo on Fri, 21 Sep 2012 18:32:17 GMT
View Forum Message <> Reply to Message

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?

> +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.

> + return name;
> +}
...
> void __init memcg_init_kmem_cache(void)
> @@ -665,6 +704,170 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
>   */
>   WARN_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
> }
> +

> +static DEFINE_MUTEX(memcg_cache_mutex);

Blank line missing. Or if it's used inside memcg_create_kmem_cache()
only move it inside the function?

> +static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
> +   struct kmem_cache *cachep)
> +{
> + struct kmem_cache *new_cachep;
> + int idx;
> +
> + BUG_ON(!memcg_can_account_kmem(memcg));

WARN_ON_ONCE() generally preferred.

> + idx = cachep->memcg_params.id;

Ah, okay so the id is assigned to the "base" cache. Maybe explain it
somewhere?

> + mutex_lock(&memcg_cache_mutex);
> + new_cachep = memcg->slabs[idx];
> + if (new_cachep)
> + goto out;
> +
> + new_cachep = kmem_cache_dup(memcg, cachep);
> +
> + if (new_cachep == NULL) {
> + new_cachep = cachep;
> + goto out;
> + }
> +
> + mem_cgroup_get(memcg);
> + memcg->slabs[idx] = new_cachep;
> + new_cachep->memcg_params.memcg = memcg;
> +out:
> + mutex_unlock(&memcg_cache_mutex);
> + return new_cachep;
> +}
> +
> +struct create_work {
> + struct mem_cgroup *memcg;
> + struct kmem_cache *cachep;
> + struct list_head list;
> +};
> +
> +/* Use a single spinlock for destruction and creation, not a frequent op */
> +static DEFINE_SPINLOCK(cache_queue_lock);

```
> +static LIST_HEAD(create_queue);
> +
> +/*
> + * Flush the queue of kmem_caches to create, because we're creating a cgroup.
> + *
> + * We might end up flushing other cgroups' creation requests as well, but
> + * they will just get queued again next time someone tries to make a slab
> + * allocation for them.
> + */
> +void memcg_flush_cache_create_queue(void)
> +{
...
> +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.

```
> +/*
> + * 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?

Thanks.

--
tejun