Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by Glauber Costa on Wed, 02 May 2012 15:34:44 GMT
View Forum Message <> Reply to Message

On 04/30/2012 05:56 PM, Suleiman Souhlal wrote:
>> +
>> +static void kmem_cache_destroy_work_func(struct work_struct *w)
>> +{
>> +	struct kmem_cache *cachep;
>> +	char *name;
>> +
>> +	spin_lock_irq(&cache_queue_lock);
>> +	while (!list_empty(&destroyed_caches)) {
>> +		cachep = container_of(list_first_entry(&destroyed_caches,
>> +		 struct mem_cgroup_cache_params, destroyed_list), struct
>> +		 kmem_cache, memcg_params);
>> +		name = (char *)cachep->name;
>> +		list_del(&cachep->memcg_params.destroyed_list);
>> +		spin_unlock_irq(&cache_queue_lock);
>> +		synchronize_rcu();
>
> Is this synchronize_rcu() still needed, now that we don't use RCU to
> protect memcgs from disappearing during allocation anymore?
>
> Also, should we drop the memcg reference we got in
> memcg_create_kmem_cache() here?

I have a reworked code I like better for this part.

It reads as follows:

```
static void kmem_cache_destroy_work_func(struct work_struct *w)
{
	struct kmem_cache *cachep;
	const char *name;
	struct mem_cgroup_cache_params *params, *tmp;
	unsigned long flags;
	LIST_HEAD(delete_unlocked);

	synchronize_rcu();

	spin_lock_irqsave(&cache_queue_lock, flags);
	list_for_each_entry_safe(params, tmp, &destroyed_caches,
destroyed_list) {
		cachep = container_of(params, struct kmem_cache,
memcg_params);
		list_move(&cachep->memcg_params.destroyed_list,
&delete_unlocked);
```

```
        }
        spin_unlock_irqrestore(&cache_queue_lock, flags);

        list_for_each_entry_safe(params, tmp, &delete_unlocked,
destroyed_list) {
                cachep = container_of(params, struct kmem_cache,
memcg_params);
                list_del(&cachep->memcg_params.destroyed_list);
                name = cachep->name;
                mem_cgroup_put(cachep->memcg_params.memcg);
                kmem_cache_destroy(cachep);
                kfree(name);
        }
}
```

I think having a list in stack is better because we don't need to hold & drop the spinlock, and can achieve more parallelism if multiple cpus are scheduling the destroy worker.

As you see, this version does a put() - so the answer to your question is yes.

synchronize_rcu() also gets a new meaning, in the sense that it only waits until everybody that is destroying a cache can have the chance to get their stuff into the list.

But to be honest, one of the things we need to do for the next version, is audit all the locking rules and write them down...

```
>> +static void memcg_create_cache_work_func(struct work_struct *w)
>> +{
>> +       struct kmem_cache *cachep;
>> +       struct create_work *cw;
>> +
>> +       spin_lock_irq(&cache_queue_lock);
>> +       while (!list_empty(&create_queue)) {
>> +               cw = list_first_entry(&create_queue, struct create_work, list);
>> +               list_del(&cw->list);
>> +               spin_unlock_irq(&cache_queue_lock);
>> +               cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep);
>> +               if (cachep == NULL)
>> +                       printk(KERN_ALERT
>> +                       "%s: Couldn't create memcg-cache for %s memcg %s\n",
>> +                       __func__, cw->cachep->name,
>> +                       cw->memcg->css.cgroup->dentry->d_name.name);
>
> We might need rcu_dereference() here (and hold rcu_read_lock()).
> Or we could just remove this message.
```

Don't understand this "or". Again, cache creation can still fail. This
is specially true in constrained memory situations.

```
>> +/*
>> + * Return the kmem_cache we're supposed to use for a slab allocation.
>> + * If we are in interrupt context or otherwise have an allocation that
>> + * can't fail, we return the original cache.
>> + * Otherwise, we will 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.
>> + *
>> + * This function returns with rcu_read_lock() held.
>> + */
>> +struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
>> +                                               gfp_t gfp)
>> +{
>> +       struct mem_cgroup *memcg;
>> +       int idx;
>> +
>> +       gfp |=  cachep->allocflags;
>> +
>> +       if ((current->mm == NULL))
>> +               return cachep;
>> +
>> +       if (cachep->memcg_params.memcg)
>> +               return cachep;
>> +
>> +       idx = cachep->memcg_params.id;
>> +       VM_BUG_ON(idx == -1);
>> +
>> +       memcg = mem_cgroup_from_task(current);
>> +       if (!mem_cgroup_kmem_enabled(memcg))
>> +               return cachep;
>> +
>> +       if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
>> +               memcg_create_cache_enqueue(memcg, cachep);
>> +               return cachep;
>> +       }
>> +
>> +       return rcu_dereference(memcg->slabs[idx]);
>
> Is it ok to call rcu_access_pointer() and rcu_dereference() without
> holding rcu_read_lock()?
```

No, but mem_cgroup_from_task should be called with rcu_read_lock() held
as well.

I forgot to change it in the comments, but this function should be
called with the rcu_read_lock() held. I was careful enough to check the
callers, and they do. But being only human, some of them might have
escaped...

```
>> +
>> +bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
>> +{
>> +       struct mem_cgroup *memcg;
>> +       bool ret = true;
>> +
>> +       rcu_read_lock();
>> +       memcg = mem_cgroup_from_task(current);
>> +
>> +       if (!mem_cgroup_kmem_enabled(memcg))
>> +               goto out;
>> +
>> +       mem_cgroup_get(memcg);
>
> Why do we need to get a reference to the memcg for every charge?
> How will this work when deleting a memcg?
```

There are two charging functions here:

mem_cgroup_charge_slab() and mem_cgroup_charge_kmem() (the later had its
name changed in my private branch, for the next submission)

The slub allocator will draw large kmalloc allocations directly from the
page allocator, which is the case this function is designed to handle.

Since we have no cache to bill this against, we need to hold the
reference here.

```
>> +       _memcg = memcg;
>> +       ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
>> +&_memcg, may_oom);
>> +       if (ret == -ENOMEM)
>> +               return ret;
>> +       else if ((ret == -EINTR) || (ret&&  (gfp&  __GFP_NOFAIL)))  {
>> +               nofail = true;
>> +               /*
>> +                * __mem_cgroup_try_charge() chose to bypass to root due
>> +                * to OOM kill or fatal signal.
>> +                * Since our only options are to either fail the
```

```
>> +             * allocation or charge it to this cgroup, force the
>> +             * change, going above the limit if needed.
>> +             */
>> +            res_counter_charge_nofail(&memcg->res, delta,&fail_res);
>
```
> We might need to charge memsw here too.

hummm, isn't there a more automated way to do that ?

I'll take a look.

> Might need to uncharge memsw.

Here too.
