Subject: Re: [PATCH v2 07/13] memcg: Slab accounting.
Posted by Suleiman Souhlal on Tue, 13 Mar 2012 22:50:01 GMT
View Forum Message <> Reply to Message

On Sun, Mar 11, 2012 at 3:25 AM, Glauber Costa <glommer@parallels.com> wrote:
> On 03/10/2012 12:39 AM, Suleiman Souhlal wrote:
>> +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);
>> +   rcu_read_unlock();
>> +}
>
>
> Is this really needed ? After this function call in slab.c, the slab code
> itself accesses cachep a thousand times. If it could be freed, it would
> already explode today for other reasons?
> Am I missing something here?

We need this because once we drop the rcu_read_lock and go to sleep,
the memcg could get deleted, which could lead to the cachep from
getting deleted as well.

So, we need to grab a reference to the cache, to make sure that the
cache doesn't disappear from under us.

>> diff --git a/init/Kconfig b/init/Kconfig
>> index 3f42cd6..e7eb652 100644
>> --- a/init/Kconfig
>> +++ b/init/Kconfig
>> @@ -705,7 +705,7 @@ config CGROUP_MEM_RES_CTLR_SWAP_ENABLED
>>   then swapaccount=0 does the trick).
>> config CGROUP_MEM_RES_CTLR_KMEM
>>   bool "Memory Resource Controller Kernel Memory accounting
>> (EXPERIMENTAL)"
>> -   depends on CGROUP_MEM_RES_CTLR&& EXPERIMENTAL
>> +   depends on CGROUP_MEM_RES_CTLR&& EXPERIMENTAL&& !SLOB
>
> Orthogonal question: Will we ever want this (SLOB) ?

I honestly don't know why someone would want to use this and slob at
the same time.
It really doesn't seem like a required feature, in my opinion.
Especially at first.

```
>> +static struct kmem_cache *
>> +memcg_create_kmem_cache(struct mem_cgroup *memcg, struct kmem_cache
>> *cachep)
>> +{
>> +	struct kmem_cache *new_cachep;
>> +	struct dentry *dentry;
>> +	char *name;
>> +	int idx;
>> +
>> +	idx = cachep->memcg_params.id;
>> +
>> +	dentry = memcg->css.cgroup->dentry;
>> +	BUG_ON(dentry == NULL);
>> +
>> +	/* Preallocate the space for "dead" at the end */
>> +	name = kasprintf(GFP_KERNEL, "%s(%d:%s)dead",
>> +	    cachep->name, css_id(&memcg->css), dentry->d_name.name);
>> +	if (name == NULL)
>> +		return cachep;
>> +	/* Remove "dead" */
>> +	name[strlen(name) - 4] = '\0';
>> +
>> +	new_cachep = kmem_cache_create_memcg(cachep, name);
>> +
>> +	/*
>> +	 * Another CPU is creating the same cache?
>> +	 * We'll use it next time.
>> +	 */
>
> This comment is a bit misleading. Is it really the only reason
> it can fail?
>
> The impression I got is that it can also fail under the normal conditions in
> which kmem_cache_create() fails.

kmem_cache_create() isn't expected to fail often.
I wasn't making an exhaustive lists of why this condition can happen,
just what I think is the most common one is.

>> +/*
>> + * Enqueue the creation of a per-memcg kmem_cache.
>> + * Called with rcu_read_lock.
>> + */
>> +static void
>> +memcg_create_cache_enqueue(struct mem_cgroup *memcg, struct kmem_cache
>> *cachep)
>> +{
```

>> +	struct create_work *cw;
>> +	unsigned long flags;
>> +
>> +	spin_lock_irqsave(&create_queue_lock, flags);
>
> If we can sleep, why not just create the cache now?
>
> Maybe it would be better to split this in two, and create the cache if
> possible, and a worker if not possible. Then w

That's how I had it in my initial patch, but I was under the
impression that you preferred if we always kicked off the creation to
the workqueue?

Which way do you prefer?

>> @@ -1756,17 +1765,23 @@ 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 (!page) {
>> +		mem_cgroup_uncharge_slab(cachep, nr_pages * PAGE_SIZE);
>> 		return NULL;
>> +	}
>
>
>
> Can't the following happen:
>
> *) mem_cgroup_charge_slab() is the first one to touch the slab.
>   Therefore, this first one is billed to root.
> *) A slab is queued for creation.
> *) alloc_pages sleep.
> *) our workers run, and create the cache, therefore filling
>   cachep->memcg_param.memcg
> *) alloc_pages still can't allocate.
> *) uncharge tries to uncharge from cachep->memcg_param.memcg,
>   which doesn't have any charges...
>

> Unless you have a strong oposition to this, to avoid this kind of
> corner cases, we could do what I was doing in the slub:
> Allocate the page first, and then account it.
> (freeing the page if it fails).
>
> I know it is not the way it is done for the user pages, but I believe it to
> be better suited for the slab.

I don't think the situation you're describing can happen, because the
memcg caches get created and selected at the beginning of the slab
allocation, in mem_cgroup_get_kmem_cache() and not in
mem_cgroup_charge_slab(), which is much later.

Once we are in mem_cgroup_charge_slab() we know that the allocation
will be charged to the cgroup.

>> @@ -2269,10 +2288,12 @@ kmem_cache_create (const char *name, size_t size,
>> size_t align,
>> }
>>
>> if (!strcmp(pc->name, name)) {
>> - printk(KERN_ERR
>> - "kmem_cache_create: duplicate cache %s\n",
>> name);
>> - dump_stack();
>> - goto oops;
>> + if (!memcg) {
>> + printk(KERN_ERR "kmem_cache_create:
>> duplicate"
>> + " cache %s\n", name);
>> + dump_stack();
>> + goto oops;
>> + }
>
> Why? Since we are apending the memcg name at the end anyway, duplicates
> still aren't expected.

Duplicates can happen if you have hierarchies, because we're only
appending the basename of the cgroup.

>> @@ -2703,12 +2787,74 @@ void kmem_cache_destroy(struct kmem_cache *cachep)
>> if (unlikely(cachep->flags& SLAB_DESTROY_BY_RCU))
>>
>> rcu_barrier();
>>
>> +#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + /* Not a memcg cache */
>> + if (cachep->memcg_params.id != -1) {

>> +		__clear_bit(cachep->memcg_params.id, cache_types);
>> +		mem_cgroup_flush_cache_create_queue();
>> +	}
>> +#endif
>
>
> This will clear the id when a leaf cache is destroyed. It seems it is not
> what we want, right? We want this id to be cleared only when
> the parent cache is gone.

id != -1, for parent caches (that's what the comment is trying to point out).
I will improve the comment.

>> +static void
>> +kmem_cache_destroy_work_func(struct work_struct *w)
>> +{
>> +	struct kmem_cache *cachep;
>> +	char *name;
>> +
>> +	spin_lock_irq(&destroy_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(&destroy_lock);
>> +		synchronize_rcu();
>> +		kmem_cache_destroy(cachep);
>
> 	^^^^^^
> 	will destroy the id.

See my previous comment.

>> @@ -3866,9 +4030,35 @@ 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);
>> +                VM_BUG_ON(actual_cachep->memcg_params.orig_cache
>> !=
>> +                   cachep);
>> +                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);
>
> 1) Another obvious candidate to be wrapped by static_branch()...
> 2) I don't trully follow why we need those references here. Can you
>   give us an example of a situation in which the cache can go away?
>
> Also note that we are making a function that used to operate mostly on
> local data now issue two atomic operations.
```

Yes, improving this is in my v3 TODO already.

The situation is very simple, and will happen every time we are freeing the last object of a dead cache.
When we free the last object, kmem_freepages() will drop the last reference, which will cause the kmem_cache to be destroyed right there.
Grabbing an additional reference before freeing the page is just a hack to avoid this situation.

It might be possible to just wrap the free path in rcu_read_lock(), or if that isn't enough, to delay the destruction until the end. I still have to think about this a bit more, to be sure.

Thanks for the detailed review,
-- Suleiman