Subject: Re: [PATCH v2 07/13] memcg: Slab accounting.
Posted by Glauber Costa on Wed, 14 Mar 2012 10:47:59 GMT
View Forum Message <> Reply to Message

On 03/14/2012 02:50 AM, Suleiman Souhlal wrote:
> 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.

Don't we grab a memcg reference when we fire the cache creation?
(I did that for slub, can't really recall from the top of my head if
you are doing it as well)

That would prevent the memcg to go away, while relieving us from the
need to take a temporary reference for every page while sleeping.

```
>>> 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.
```

Agree. It was more a question to see if anyone would speak up for it.
But certainly not me.

```
>>> +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.

Keep in mind that our notion of "fail often" may start to change when
we start limiting the amount of kernel memory =p.

Specially in nested cgroups limited by its parent.

So apart from the comment issue, the problem here to me seems to be that:

yes, kmem_cache_create failing is rare. But the circumstances in which
it can happen all involve memory pressure. And in this case, we'll leave
memcg->slabs[idx] as NULL, which means we'll keep trying to create the
cache in further allocations.

This seems at best a tricky way to escape the memcg constraint...

I am not sure this is the behavior we want. Have to think a little bit.

```
>
>>> +/*
>>> + * 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?

Sorry If I misled you. But what I remember mentioning, was that it was
maybe better to create some of the caches right away, instead of putting
the into the workqueue at all. So earlier, rather than later.

That said, how I view this particular issue changed quite a bit over the
past days, due to our discussions. Specially, see the last mail I wrote
to Kame as reply to your patchset. I think that queuing up stuff
in the workqueue may get quite handy in the end.

But in the interest of having  less objects scaping memcg, how about we
call cond_resched() when we can sleep, after we kicked out the worker?

This way we don't need to deal with conditionals for can sleep vs can't
sleep, (simpler code), while having the cache created right away when it
can.

```
>>> @@ -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.

That's not how I read it. Since there is no completion guarantees coming
from the workqueue, I really don't see how we can be sure that the data
in cachep->memcg_param.memcg won't change.

You are right that touching the slab actually happens in
mem_cgroup_get_kmem_cache(). That is called in kmem_cache_aloc(). And
the first object is likely to be billed to the parent cgroup (or root)

Now imagine that cache being full, so we need a new page for it.
This will quickly lead us to cache_grow(), and all the other steps are
therefore the same.

So how can we guarantee that the memcg pointer is stable between alloc
and free?

```
>>> @@ -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.
No, we're appending the css id now too, precisely to cope with the
duplicates problem.

>
>>> @@ -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.

/me goes check all the code again...

Does that mean that when two memcg's are creating the same cache they
will end up with different ids??

---

## Subject: Re: [PATCH v2 07/13] memcg: Slab accounting.
## Posted by Suleiman Souhlal on Wed, 14 Mar 2012 22:04:33 GMT

View Forum Message <> Reply to Message

On Wed, Mar 14, 2012 at 3:47 AM, Glauber Costa <glommer@parallels.com> wrote:
> On 03/14/2012 02:50 AM, Suleiman Souhlal wrote:
>>
>> 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.
>
>
> Don't we grab a memcg reference when we fire the cache creation?
> (I did that for slub, can't really recall from the top of my head if
> you are doing it as well)
>
> That would prevent the memcg to go away, while relieving us from the
> need to take a temporary reference for every page while sleeping.

The problem isn't the memcg going away, but the cache going away.

```
>>>> +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.
>
>
> Keep in mind that our notion of "fail often" may start to change when
> we start limiting the amount of kernel memory =p.
>
> Specially in nested cgroups limited by its parent.
>
> So apart from the comment issue, the problem here to me seems to be that:
>
> yes, kmem_cache_create failing is rare. But the circumstances in which it
> can happen all involve memory pressure. And in this case, we'll leave
> memcg->slabs[idx] as NULL, which means we'll keep trying to create the cache
> in further allocations.
>
> This seems at best a tricky way to escape the memcg constraint...
>
> I am not sure this is the behavior we want. Have to think a little bit.

Keep in mind that this function is only called in workqueue context.
(In the earlier revision of the patchset this function was called in

the process context, but kmem_cache_create() would ignore memory
limits, because of __GFP_NOACCOUNT.)

```
>>>> @@ -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.
>
>
> That's not how I read it. Since there is no completion guarantees coming
> from the workqueue, I really don't see how we can be sure that the data in
> cachep->memcg_param.memcg won't change.
>
> You are right that touching the slab actually happens in
> mem_cgroup_get_kmem_cache(). That is called in kmem_cache_aloc(). And the
> first object is likely to be billed to the parent cgroup (or root)
>
> Now imagine that cache being full, so we need a new page for it.
> This will quickly lead us to cache_grow(), and all the other steps are
> therefore the same.
>
> So how can we guarantee that the memcg pointer is stable between alloc and
> free?

When mem_cgroup_get_kmem_cache() returns a memcg cache, that cache has
already been created.

The memcg pointer is not stable between alloc and free: It can become
NULL when the cgroup gets deleted, at which point the accounting has
been "moved to root" (uncharged from the cgroup it was charged in).
When that has happened, we don't want to uncharge it again.
I think the current code already handles this situation.

```
>>>> @@ -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.
>
>
> /me goes check all the code again...
>
> Does that mean that when two memcg's are creating the same cache they will
> end up with different ids??

No, only parent caches have an id that is not -1. memcg caches always
have an id of -1.
Sorry if that wasn't clear. I will try to document it better.

-- Suleiman