Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Thu, 16 Aug 2012 03:37:01 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15 2012, Glauber Costa wrote:

> On 08/15/2012 09:12 PM, Greg Thelen wrote:
>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>
>>> On 08/15/2012 08:38 PM, Greg Thelen wrote:
>>>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>>>
>>>>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>>>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>>>>
>>>>>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>>>>>> + mem_cgroup_put(memcg);
>>>>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>>>>> reference ?
>>>>>>> This is of course the pair of the mem_cgroup_get() you commented on
>>>>>>> earlier. If we need one, we need the other. If we don't need one, we
>>>>>>> don't need the other =)
>>>>>>>
>>>>>>> The guarantee we're trying to give here is that the memcg structure will
>>>>>>> stay around while there are dangling charges to kmem, that we decided
>>>>>>> not to move (remember: moving it for the stack is simple, for the slab
>>>>>>> is very complicated and ill-defined, and I believe it is better to treat
>>>>>>> all kmem equally here)
>>>>>>
>>>>>> By keeping memcg structures hanging around until the last referring kmem
>>>>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>>>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>>>>> this would prevent creation of new cgroups until these zombies are
>>>>>> dereferenced.
>>>>>
>>>>> Yes, but although this patch makes it more likely, it doesn't introduce
>>>>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>>>>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>>>>> around.
>>>>
>>>> Fair point. But this doesn't seems like a feature. It's probably not
>>>> needed initially, but what do you think about creating a
>>>> memcg_kernel_context structure which is allocated when memcg is
>>>> allocated? Kernel pages charged to a memcg would have
>>>> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
>>>> would allow the mem_cgroup and its css_id to be deleted when the cgroup

>>>> is unlinked from cgroupfs while allowing for the active kernel pages to
>>>> continue pointing to a valid memcg_kernel_context. This would be a
>>>> reference counted structure much like you are doing with memcg. When a
>>>> memcg is deleted the memcg_kernel_context would be linked into its
>>>> surviving parent memcg. This would avoid needing to visit each kernel
>>>> page.
>>>
>>> You need more, you need at the res_counters to stay around as well. And
>>> probably other fields.
>>
>> I am not sure the res_counters would need to stay around. Once a
>> memcg_kernel_context has been reparented, then any future kernel page
>> uncharge calls will uncharge the parent res_counter.
>
> Well, if you hold the memcg due to a reference, like in the dentry case,
> then fine. But if this is a dangling charge, as will be the case with
> the slab, then you have to uncharge it.
>
> An arbitrary number of parents might have been deleted as well, so you
> need to transverse them all until you reach a live parent to uncharge from.

I was thinking that each time a memcg is deleted move the
memcg_kernel_context from the victim memcg to its parent. When moving,
also update the context to refer to the parent and link context to
parent:

```
  for_each_kernel_context(kernel_context, memcg) {
    kernel_context->memcg = memcg->parent;
    list_add(&kernel_context->list, &memcg->parent->kernel_contexts);
  }
```

Whenever pages referring to a memcg_kernel_context are uncharged they
will uncharge the nearest surviving parent memcg.

> To do that, your counters have to be still alive.

The counters of nearest surviving parent will be alive and pointed to by
memcg_kernel_context->memcg.

>>> So my fear here is that as you add fields to that structure, you can
>>> defeat a bit the goal of reducing memory consumption. Still leaves the
>>> css space, yes. But by doing this we can introduce some subtle bugs by
>>> having a field in the wrong structure.
>>>
>>> Did you observe that to be a big problem in your systems?
>>
>> No I have not seen this yet. But our past solutions have reparented
>> kmem_cache's to root memcg so we have been avoiding zombie memcg. My
>> concerns with your approach are just a suspicion because we have been

>> experimenting with accounting of even more kernel memory (e.g. vmalloc,
>> kernel stacks, page tables). As the scope of such accounting grows the
>> chance of long lived charged pages grows and thus the chance of zombies
>> which exhaust the css_id space grows.
>
> Well, since we agree this can all be done under the hood, I'd say let's
> wait until a problem actually exists, since the solution is likely to be
> a bit convoluted...
>
> I personally believe that if won't have a lot of task movement, most of
> the data will go away as the cgroup dies. The remainder shouldn't be too
> much to hold it in memory for a lot of time. This is of course assuming
> a real use case, not an adversarial scenario, which is quite easy to
> come up with: just create a task, hold a bunch of kmem, move the task
> away, delete the cgroup, etc.
>
> That said, nothing stops us to actively try to create a scenario that
> would demonstrate such a problem.

With our in-house per-memcg slab accounting (similar to what's discussed
here), we're seeing a few slab allocations (mostly radix_tree_node) that
survive a long time after memcg deletion. This isn't meant as criticism
of this patch series, just an fyi that I expect there will be scenarios
where some dead kmem caches will live for a long time. Though I think
that in your patches a dead kmem cache does not hold reference to the
memcg.