Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Mel Gorman on Wed, 15 Aug 2012 13:22:44 GMT
View Forum Message <> Reply to Message

```
On Wed, Aug 15, 2012 at 01:08:08PM +0400, Glauber Costa wrote:
> On 08/14/2012 07:16 PM, Mel Gorman wrote:
> > On Thu, Aug 09, 2012 at 05:01:15PM +0400, Glauber Costa wrote:
> >> When a process tries to allocate a page with the __GFP_KMEMCG flag, the
> >> page allocator will call the corresponding memcg functions to validate
> >> the allocation. Tasks in the root memcg can always proceed.
> >>
> >> To avoid adding markers to the page - and a kmem flag that would
> >> necessarily follow, as much as doing page_cgroup lookups for no reason,
> >
> > As you already guessed, doing a page_cgroup in the page allocator free
> > path would be a no-go.
>
> Specifically yes, but in general, you will be able to observe that I am
> taking all the possible measures to make sure existing paths are
> disturbed as little as possible.
>
> Thanks for your review here
>
> >>
> >> diff --git a/mm/page_alloc.c b/mm/page_alloc.c
> >> index b956cec..da341dc 100644
> >> --- a/mm/page_alloc.c
> >> +++ b/mm/page_alloc.c
> >> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
> >>   struct page *page = NULL;
> >>   int migratetype = allocflags_to_migratetype(gfp_mask);
> >>   unsigned int cpuset_mems_cookie;
> >> + void *handle = NULL;
> >>
> >>   gfp_mask &= gfp_allowed_mask;
> >>
> >> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
> >>    return NULL;
> >>
> >>   /*
> >> +  * Will only have any effect when __GFP_KMEMCG is set.
> >> +  * This is verified in the (always inline) callee
> >> +  */
> >> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
> >
> > memcg_kmem_new_page takes a void * parameter already but here you are
> > passing in a void **. This probably happens to work because you do this
> >
```

> > struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
> >
> > but that appears to defeat the purpose of having an opaque type as a
> > "handle". You have to treat it different then passing it into the commit
> > function because it expects a void *. The motivation for an opaque type
> > is completely unclear to me and how it is managed with a mix of void *
> > and void ** is very confusing.
>
> okay.
>
> The opaque exists because I am doing speculative charging.

I do not get why speculative charing would mandate an opaque type or
"handle". It looks like like a fairly standard prepare/commit pattern to me.

> I believe it
> to be a better and less complicated approach then letting a page appear
> and then charging it. Besides being consistent with the rest of memcg,
> it won't create unnecessary disturbance in the page allocator
> when the allocation is to fail.
>

I still don't get why you did not just return a mem_cgroup instead of a
handle.

> Now, tasks can move between memcgs, so we can't rely on grabbing it from
> current in commit_page, so we pass it around as a handle.

You could just as easily passed around the mem_cgroup and it would have
been less obscure. Maybe this makes sense from a memcg context and matches
some coding pattern there that I'm not aware of.

> Also, even if
> the task could not move, we already got it once from the task, and that
> is not for free. Better save it.
>
> Aside from the handle needed, the cost is more or less the same compared
> to doing it in one pass. All we do by using speculative charging is to
> split the cost in two, and doing it from two places.
> We'd have to charge + update page_cgroup anyway.
>
> As for the type, do you think using struct mem_cgroup would be less
> confusing?
>

Yes and returning the mem_cgroup or NULL instead of bool.

> > On a similar note I spotted #define memcg_kmem_on 1 . That is also

> > different just for the sake of it. The convension is to do something
> > like this
> >
> > /* This helps us to avoid #ifdef CONFIG_NUMA */
> > #ifdef CONFIG_NUMA
> > #define NUMA_BUILD 1
> > #else
> > #define NUMA_BUILD 0
> > #endif
>
> For simple defines, yes. But a later patch will turn this into a static
> branch test. memcg_kmem_on will be always 0 when compile-disabled, but
> when enable will expand to static_branch(&...).
>

I see.

>
> > memcg_kmem_on was difficult to guess based on its name. I thought initially
> > that it would only be active if a memcg existed or at least something like
> > mem_cgroup_disabled() but it's actually enabled if CONFIG_MEMCG_KMEM is set.
>
> For now. And I thought that adding the static branch in this patch would
> only confuse matters.

Ah, I see now. I had stopped reading the series once I reached this
patch. I don't think it would have mattered much to collapse the two
patches together but ok.

The static key handling does look a little suspicious. You appear to do
reference counting in memcg_update_kmem_limit for every mem_cgroup_write()
but decrement it on memcg exit. This does not appear as if it would be
symmetric if the memcg files were written to multiple times (maybe that's
not allowed?). Either way, the comment says it can never be disabled but
as you have static_key_slow_dec calls it would appear that you *do*
support them being disabled. Confusing.

> The placeholder is there, but it is later patched
> to the final thing.
> With that explained, if you want me to change it to something else, I
> can do it. Should I ?
>

Not in this patch anyway. I would have preferred a pattern like this but
that's about it.

#ifdef CONFIG_MEMCG_KMEM
extern struct static_key memcg_kmem_enabled_key;

```
static inline int memcg_kmem_enabled(void)
{
	return static_key_false(&memcg_kmem_enabled_key);
}
#else

static inline bool memcg_kmem_enabled(void)
{
	return false;
}
#endif
```

Two reasons. One, it does not use the terms "on" and "enabled"
interchangeably. The other reason is down to taste as I'm copying the
pattern I used myself for sk_memalloc_socks(). Of course I am biased.

Also, why is the key exported?

> > I also find it *very* strange to have a function named as if it is an
> > allocation-style function when it in fact it's looking up a mem_cgroup
> > and charging it (and uncharging it in the error path if necessary). If
> > it was called memcg_kmem_newpage_charge I might have found it a little
> > better.
>
> I don't feel strongly about names in general. I can change it.
> Will update to memcg_kmem_newpage_charge() and memcg_kmem_page_uncharge().
>

I would prefer that anyway. Names have meaning and people make assumptions on
the implementation depending on the name. We should try to be as consistent
as possible or maintenance becomes harder. I know there are areas where
we are not consistent at all but we should not compound the problem.

> > This whole operation also looks very expensive (cgroup lookups, RCU locks
> > taken etc) but I guess you're willing to take that cost in the same of
> > isolating containers from each other. However, I strongly suggest that
> > this overhead is measured in advance. It should not stop the series being
> > merged as such but it should be understood because if the cost is high
> > then this feature will be avoided like the plague. I am skeptical that
> > distributions would enable this by default, at least not without support
> > for cgroup_disable=kmem
>
> Enabling this feature will bring you nothing, therefore, no (or little)
> overhead. Nothing of this will be patched in until the first memcg gets
> kmem limited. The mere fact of moving tasks to memcgs won't trigger any
> of this.
>

ok.

> I haven't measured this series in particular, but I did measure the slab
> series (which builds ontop of this). I found the per-allocation cost to
> be in the order of 2-3 % for tasks living in limited memcgs, and
> hard to observe when living in the root memcg (compared of course to the
> case of a task running on root memcg without those patches)
>

Depending on the workload that 2-3% could be a lot but at least you're
aware of it.

> I also believe the folks from google also measured this. They may be
> able to spit out numbers grabbed from a system bigger than mine =p
>
> > As this thing is called from within the allocator, it's not clear why
> > __memcg_kmem_new_page is exported. I can't imagine why a module would call
> > it directly although maybe you cover that somewhere else in the series.
>
> Okay, more people commented on this, so let me clarify: They shouldn't
> be. They were initially exported when this was about the slab only,
> because they could be called from inlined functions from the allocators.
> Now that the charge/uncharge was moved to the page allocator - which
> already allowed me the big benefit of separating this in two pieces,
> none of this needs to be exported.
>
> Sorry for not noticing this myself, but thanks for the eyes =)
>

You're welcome. I expect to see all the exports disappear so. If there
are any exports left I think it would be important to document why they
have to be exported. This is particularly true because they are
EXPORT_SYMBOL not EXPORT_SYMBOL_GPL. I think it would be good to know in
advance why a module (particularly an out-of-tree one) would be
interested.

> > From the point of view of a hook, that is acceptable but just barely. I have
> > slammed other hooks because it was possible for a subsystem to override them
> > meaning the runtime cost could be anything. I did not spot a similar issue
> > here but if I missed it, it's still unacceptable. At least here the cost
> > is sortof predictable and only affects memcg because of the __GFP_KMEMCG
> > check in memcg_kmem_new_page.
>
> Yes, that is the idea. And I don't think anyone should override those,
> so I don't see them as hooks in this sense.
>

Indeed not, callbacks are the real issue.

```
> >> +  return NULL;
> >> +
> >> + /*
> >>    * Check the zones suitable for the gfp_mask contain at least one
> >>    * valid zone. It's possible to have an empty zonelist as a result
> >>    * of GFP_THISNODE and a memoryless node
> >> @@ -2583,6 +2591,8 @@ out:
> >>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
> >>    goto retry_cpuset;
> >>
> >> + memcg_kmem_commit_page(page, handle, order);
> >> +
> >
> > As a side note, I'm not keen on how you shortcut these functions. They
> > are all function calls because memcg_kmem_commit_page() will always call
> > __memcg_kmem_commit_page() to check the handle once it's compiled in.
> > The handle==NULL check should have happened in the inline function to save
> > a few cycles.
> >
>
> It is already happening on my updated series after a comment from Kame
> pointed this out.
>
```

ok.

```
> > <SNIP>
> >
> > memcg_kmem_new_page makes the following check
> >
> > +       if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> > +               return true;
> >
> > so if the allocation had __GFP_NOFAIL, it does not get charged but can
> > still be freed. I didn't check if this is really the case but it looks
> > very suspicious.
>
> No, it can't be freed (uncharged), because in that case, we won't fill
> in the memcg information in page cgroup.
>
```

Ah, I see.

```
> > Again, this is a fairly heavy operation.
>
>
> Mel, once I address all the issues you pointed out here, do you think
```

> this would be in an acceptable state for merging? Do you still have any
> fundamental opposition to this?
>

I do not have a fundamental opposition to it, particularly as it only
has an impact when it's enabled. This is not an ack either though as I
see the series in general still has a lot of feedback outstanding
including this patch.

--
Mel Gorman
SUSE Labs