Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Mel Gorman on Tue, 14 Aug 2012 15:16:16 GMT
View Forum Message <> Reply to Message

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.

This is my first time glancing at the series and I'm only paying close
attention to this patch so pardon me if my observations have been made
already.

```
> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
> for telling the page allocator that this is such an allocation at
> free_pages() time. This is done by the invocation of
> __free_accounted_pages() and free_accounted_pages().
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
>  include/linux/gfp.h |  3 +++
>  mm/page_alloc.c     | 38 ++++++++++++++++++++++++++++++++++++++
>  2 files changed, 41 insertions(+)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index d8eae4d..029570f 100644
> --- a/include/linux/gfp.h
> +++ b/include/linux/gfp.h
> @@ -370,6 +370,9 @@ extern void free_pages(unsigned long addr, unsigned int order);
>  extern void free_hot_cold_page(struct page *page, int cold);
>  extern void free_hot_cold_page_list(struct list_head *list, int cold);
>
> +extern void __free_accounted_pages(struct page *page, unsigned int order);
> +extern void free_accounted_pages(unsigned long addr, unsigned int order);
> +
>  #define __free_page(page) __free_pages((page), 0)
>  #define free_page(addr) free_pages((addr), 0)
>
> 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.

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
```

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.

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. While I believe you have to take care to avoid confusion with
mem_cgroup_newpage_charge, it would be preferable if the APIs were similar.
memcg is hard enough as it is to understand without having different APIs.

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

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.

>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.

```
> +  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.

This also has the feel that the call of memcg_kmem_commit_page belongs in
prep_new_page() but I recognise that requires passing the opaque handler
around which would be very ugly.

```
>   return page;
>  }
>  EXPORT_SYMBOL(__alloc_pages_nodemask);
> @@ -2635,6 +2645,34 @@ void free_pages(unsigned long addr, unsigned int order)
>
>  EXPORT_SYMBOL(free_pages);
>
> +/*
> + * __free_accounted_pages and free_accounted_pages will free pages allocated
> + * with __GFP_KMEMCG.
> + *
> + * Those pages are accounted to a particular memcg, embedded in the
> + * corresponding page_cgroup. To avoid adding a hit in the allocator to search
> + * for that information only to find out that it is NULL for users who have no
> + * interest in that whatsoever, we provide these functions.
> + *
> + * The caller knows better which flags it relies on.
> + */
> +void __free_accounted_pages(struct page *page, unsigned int order)
> +{
> + memcg_kmem_free_page(page, order);
> + __free_pages(page, order);
> +}
> +EXPORT_SYMBOL(__free_accounted_pages);
```

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.

Again, this is a fairly heavy operation.

```
> +
> +void free_accounted_pages(unsigned long addr, unsigned int order)
> +{
> + if (addr != 0) {
> +  VM_BUG_ON(!virt_addr_valid((void *)addr));
> +  memcg_kmem_free_page(virt_to_page((void *)addr), order);
> +  __free_pages(virt_to_page((void *)addr), order);
> + }
> +}
> +EXPORT_SYMBOL(free_accounted_pages);
> +
>  static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
>  {
>   if (addr) {
```

--
Mel Gorman
SUSE Labs