Subject: Re: [PATCH 03/10] memcg: infrastructure to match an allocation to the right cache
Posted by Kirill A. Shutsemov on Wed, 25 Jul 2012 21:53:32 GMT
View Forum Message <> Reply to Message

On Wed, Jul 25, 2012 at 06:38:14PM +0400, Glauber Costa wrote:
> The page allocator is able to bind a page to a memcg when it is
> allocated. But for the caches, we'd like to have as many objects as
> possible in a page belonging to the same cache.
>
> This is done in this patch by calling memcg_kmem_get_cache in the
> beginning of every allocation function. This routing is patched out by
> static branches when kernel memory controller is not being used.
>
> It assumes that the task allocating, which determines the memcg in the
> page allocator, belongs to the same cgroup throughout the whole process.
> Misacounting can happen if the task calls memcg_kmem_get_cache() while
> belonging to a cgroup, and later on changes. This is considered
> acceptable, and should only happen upon task migration.
>
> Before the cache is created by the memcg core, there is also a possible
> imbalance: the task belongs to a memcg, but the cache being allocated
> from is the global cache, since the child cache is not yet guaranteed to
> be ready. This case is also fine, since in this case the GFP_KMEMCG will
> not be passed and the page allocator will not attempt any cgroup
> accounting.
>
> 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/memcontrol.h | 38 ++++++++
> init/Kconfig | 2 +-
> mm/memcontrol.c | 221 ++-
> 3 files changed, 259 insertions(+), 2 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index d9229a3..bd1f34b 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -423,6 +423,8 @@ int memcg_css_id(struct mem_cgroup *memcg);
> void memcg_register_cache(struct mem_cgroup *memcg,
>   struct kmem_cache *s);
> void memcg_release_cache(struct kmem_cache *cachep);

> +struct kmem_cache *
> +__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp);
> #else
> static inline void memcg_register_cache(struct mem_cgroup *memcg,
> struct kmem_cache *s)
> @@ -456,6 +458,12 @@ __memcg_kmem_commit_page(struct page *page, struct
mem_cgroup *handle,
> int order)
> {
> }
> +
> +static inline struct kmem_cache *
> +__memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{
> + return cachep;
> +}
> #endif /* CONFIG_MEMCG_KMEM */
>
> /**
> @@ -515,5 +523,35 @@ void memcg_kmem_commit_page(struct page *page, struct
mem_cgroup *handle,
> if (memcg_kmem_on)
> __memcg_kmem_commit_page(page, handle, order);
> }
> +
> +/**
> + * memcg_kmem_get_kmem_cache: selects the correct per-memcg cache for allocation
> + * @cachep: the original global kmem cache
> + * @gfp: allocation flags.
> + *
> + * This function assumes that the task allocating, which determines the memcg
> + * in the page allocator, belongs to the same cgroup throughout the whole
> + * process. Misacounting can happen if the task calls memcg_kmem_get_cache()
> + * while belonging to a cgroup, and later on changes. This is considered
> + * acceptable, and should only happen upon task migration.
> + *
> + * Before the cache is created by the memcg core, there is also a possible
> + * imbalance: the task belongs to a memcg, but the cache being allocated from
> + * is the global cache, since the child cache is not yet guaranteed to be
> + * ready. This case is also fine, since in this case the GFP_KMEMCG will not be
> + * passed and the page allocator will not attempt any cgroup accounting.
> + */
> +static __always_inline struct kmem_cache *
> +memcg_kmem_get_cache(struct kmem_cache *cachep, gfp_t gfp)
> +{
> + if (!memcg_kmem_on)
> + return cachep;
> + if (gfp & __GFP_NOFAIL)

> + return cachep;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> + return cachep;
> +
> + return __memcg_kmem_get_cache(cachep, gfp);
> +}
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/init/Kconfig b/init/Kconfig
> index 547bd10..610cfd3 100644
> --- a/init/Kconfig
> +++ b/init/Kconfig
> @@ -741,7 +741,7 @@ config MEMCG_SWAP_ENABLED
>   then swapaccount=0 does the trick).
> config MEMCG_KMEM
>   bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
> - depends on MEMCG && EXPERIMENTAL
> + depends on MEMCG && EXPERIMENTAL && !SLOB
>   default n
>   help
>   The Kernel Memory extension for Memory Resource Controller can limit
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 88bb826..8d012c7 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -14,6 +14,10 @@
>   * Copyright (C) 2012 Parallels Inc. and Google Inc.
>   * Authors: Glauber Costa and Suleiman Souhlal
>   *
> + * Kernel Memory Controller
> + * Copyright (C) 2012 Parallels Inc. and Google Inc.
> + * Authors: Glauber Costa and Suleiman Souhlal
> + *
>   * This program is free software; you can redistribute it and/or modify
>   * it under the terms of the GNU General Public License as published by
>   * the Free Software Foundation; either version 2 of the License, or
> @@ -339,6 +343,11 @@ struct mem_cgroup {
> #ifdef CONFIG_INET
>   struct tcp_memcontrol tcp_mem;
> #endif
> +
> +#ifdef CONFIG_MEMCG_KMEM
> + /* Slab accounting */
> + struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
> +#endif
> };
>
> enum {

```
> @@ -532,6 +541,40 @@ static inline bool memcg_kmem_enabled(struct mem_cgroup
*memcg)
>     memcg->kmem_accounted;
>  }
>
> +static char *memcg_cache_name(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> +{
> + char *name;
> + struct dentry *dentry;
> +
> + rcu_read_lock();
> + dentry = rcu_dereference(memcg->css.cgroup->dentry);
> + rcu_read_unlock();
> +
> + BUG_ON(dentry == NULL);
> +
> + name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
> +      cachep->name, css_id(&memcg->css), dentry->d_name.name);
> +
> + return name;
> +}
> +
> +static struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
> +       struct kmem_cache *s)
> +{
> + char *name;
> + struct kmem_cache *new;
> +
> + name = memcg_cache_name(memcg, s);
> + if (!name)
> +  return NULL;
> +
> + new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
> +           (s->flags & ~SLAB_PANIC), s->ctor);
> +
> + kfree(name);
> + return new;
> +}
> +
>  struct ida cache_types;
>
>  void memcg_register_cache(struct mem_cgroup *memcg, struct kmem_cache *cachep)
> @@ -656,6 +699,14 @@ void __memcg_kmem_free_page(struct page *page, int order)
>  }
>  EXPORT_SYMBOL(__memcg_kmem_free_page);
>
> +static void memcg_slab_init(struct mem_cgroup *memcg)
> +{
```

> + int i;
> +
> + for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
> + memcg->slabs[i] = NULL;
> +}

It seems redundant. mem_cgroup_alloc() uses kzalloc()/vzalloc() to
allocate struct mem_cgroup.

--
 Kirill A. Shutemov