
Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure
Posted by [Suleiman Souhlal](#) on Mon, 30 Apr 2012 20:56:46 GMT
[View Forum Message](#) <> [Reply to Message](#)

On Sun, Apr 22, 2012 at 4:53 PM, Glauber Costa <glommer@parallels.com> wrote:

> With all the dependencies already in place, this patch introduces
> the charge/uncharge functions for the slab cache accounting in memcg.
>
> Before we can charge a cache, we need to select the right cache.
> This is done by using the function `__mem_cgroup_get_kmem_cache()`.
>
> If we should use the root kmem cache, this function tries to detect
> that and return as early as possible.
>
> The charge and uncharge functions comes in two flavours:
> * `__mem_cgroup_(un)charge_slab()`, that assumes the allocation is
> a slab page, and
> * `__mem_cgroup_(un)charge_kmem()`, that does not. This later exists
> because the slub allocator draws the larger kmallocc allocations
> from the page allocator.
>
> In memcontrol.h those functions are wrapped in inline accessors.
> The idea is to later on, patch those with jump labels, so we don't
> incur any overhead when no mem cgroups are being used.
>
> Because the slub allocator tends to inline the allocations whenever
> it can, those functions need to be exported so modules can make use
> of it properly.
>
> I apologize in advance to the reviewers. This patch is quite big, but
> I was not able to split it any further due to all the dependencies
> between the code.
>
> This code is inspired by the code written by Suleiman Souhlal,
> but heavily changed.
>
> 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 | 68 +++++++
> init/Kconfig | 2 +-
> mm/memcontrol.c | 373
+++++

```

> 3 files changed, 441 insertions(+), 2 deletions(-)
>
> +
> +static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
> +                                                    struct kmem_cache
> +                                                    *cachep)
> +{
> +    struct kmem_cache *new_cachep;
> +    int idx;
> +
> +    BUG_ON(!mem_cgroup_kmem_enabled(memcg));
> +
> +    idx = cachep->memcg_params.id;
> +
> +    mutex_lock(&memcg_cache_mutex);
> +    new_cachep = memcg->slabs[idx];
> +    if (new_cachep)
> +        goto out;
> +
> +    new_cachep = kmem_cache_dup(memcg, cachep);
> +
> +    if (new_cachep == NULL) {
> +        new_cachep = cachep;
> +        goto out;
> +    }
> +
> +    mem_cgroup_get(memcg);
> +    memcg->slabs[idx] = new_cachep;
> +    new_cachep->memcg_params.memcg = memcg;
> +out:
> +    mutex_unlock(&memcg_cache_mutex);
> +    return new_cachep;
> +}
> +
> +struct create_work {
> +    struct mem_cgroup *memcg;
> +    struct kmem_cache *cachep;
> +    struct list_head list;
> +};
> +
> +/* Use a single spinlock for destruction and creation, not a frequent op */
> +static DEFINE_SPINLOCK(cache_queue_lock);
> +static LIST_HEAD(create_queue);
> +static LIST_HEAD(destroyed_caches);
> +
> +static void kmem_cache_destroy_work_func(struct work_struct *w)
> +{

```

```

> + struct kmem_cache *cachep;
> + char *name;
> +
> + spin_lock_irq(&cache_queue_lock);
> + while (!list_empty(&destroyed_caches)) {
> +     cachep = container_of(list_first_entry(&destroyed_caches,
> +     struct mem_cgroup_cache_params, destroyed_list), struct
> +     kmem_cache, memcg_params);
> +     name = (char *)cachep->name;
> +     list_del(&cachep->memcg_params.destroyed_list);
> +     spin_unlock_irq(&cache_queue_lock);
> +     synchronize_rcu();

```

Is this `synchronize_rcu()` still needed, now that we don't use RCU to protect memcgs from disappearing during allocation anymore?

Also, should we drop the memcg reference we got in `memcg_create_kmem_cache()` here?

```

> +     kmem_cache_destroy(cachep);
> +     kfree(name);
> +     spin_lock_irq(&cache_queue_lock);
> + }
> + spin_unlock_irq(&cache_queue_lock);
> +}
> +static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
> +
> +void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
> +{
> +    unsigned long flags;
> +
> +    BUG_ON(cachep->memcg_params.id != -1);
> +
> +    /*
> +     * We have to defer the actual destroying to a workqueue, because
> +     * we might currently be in a context that cannot sleep.
> +     */
> +    spin_lock_irqsave(&cache_queue_lock, flags);
> +    list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
> +    spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> +    schedule_work(&kmem_cache_destroy_work);
> +}
> +
> +
> +/*
> + * Flush the queue of kmem_caches to create, because we're creating a cgroup.
> + *

```

```

> + * We might end up flushing other cgroups' creation requests as well, but
> + * they will just get queued again next time someone tries to make a slab
> + * allocation for them.
> + */
> +void mem_cgroup_flush_cache_create_queue(void)
> +{
> +    struct create_work *cw, *tmp;
> +    unsigned long flags;
> +
> +    spin_lock_irqsave(&cache_queue_lock, flags);
> +    list_for_each_entry_safe(cw, tmp, &create_queue, list) {
> +        list_del(&cw->list);
> +        kfree(cw);
> +    }
> +    spin_unlock_irqrestore(&cache_queue_lock, flags);
> +}
> +
> +static void memcg_create_cache_work_func(struct work_struct *w)
> +{
> +    struct kmem_cache *cachep;
> +    struct create_work *cw;
> +
> +    spin_lock_irq(&cache_queue_lock);
> +    while (!list_empty(&create_queue)) {
> +        cw = list_first_entry(&create_queue, struct create_work, list);
> +        list_del(&cw->list);
> +        spin_unlock_irq(&cache_queue_lock);
> +        cachep = memcg_create_kmem_cache(cw->memcg, cw->cachep);
> +        if (cachep == NULL)
> +            printk(KERN_ALERT
> +                "%s: Couldn't create memcg-cache for %s memcg %s\n",
> +                __func__, cw->cachep->name,
> +                cw->memcg->css.cgroup->dentry->d_name.name);

```

We might need `rcu_dereference()` here (and hold `rcu_read_lock()`).
 Or we could just remove this message.

```

> +        /* Drop the reference gotten when we enqueued. */
> +        css_put(&cw->memcg->css);
> +        kfree(cw);
> +        spin_lock_irq(&cache_queue_lock);
> +    }
> +    spin_unlock_irq(&cache_queue_lock);
> +}
> +
> +static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
> +
> +/*

```

```

> + * 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(&cache_queue_lock, flags);
> +    list_for_each_entry(cw, &create_queue, list) {
> +        if (cw->memcg == memcg && cw->cachep == cachep) {
> +            spin_unlock_irqrestore(&cache_queue_lock, flags);
> +            return;
> +        }
> +    }
> +    spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> +    /* The corresponding put will be done in the workqueue. */
> +    if (!css_tryget(&memcg->css))
> +        return;
> +
> +    cw = kcalloc_no_account(sizeof(struct create_work), GFP_NOWAIT);
> +    if (cw == NULL) {
> +        css_put(&memcg->css);
> +        return;
> +    }
> +
> +    cw->memcg = memcg;
> +    cw->cachep = cachep;
> +    spin_lock_irqsave(&cache_queue_lock, flags);
> +    list_add_tail(&cw->list, &create_queue);
> +    spin_unlock_irqrestore(&cache_queue_lock, flags);
> +
> +    schedule_work(&memcg_create_cache_work);
> +}
> +
> +/*
> + * Return the kmem_cache we're supposed to use for a slab allocation.
> + * If we are in interrupt context or otherwise have an allocation that
> + * can't fail, we return the original cache.
> + * Otherwise, we will try to use the current memcg's version of the cache.
> + *
> + * If the cache does not exist yet, if we are the first user of it,
> + * we either create it immediately, if possible, or create it asynchronously
> + * in a workqueue.
> + * In the latter case, we will let the current allocation go through with
> + * the original cache.

```

```

> + *
> + * This function returns with rcu_read_lock() held.
> + */
> + struct kmem_cache * __mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
> +                                              gfp_t gfp)
> + {
> +     struct mem_cgroup *memcg;
> +     int idx;
> +
> +     gfp |= cachep->allocflags;
> +
> +     if ((current->mm == NULL))
> +         return cachep;
> +
> +     if (cachep->memcg_params.memcg)
> +         return cachep;
> +
> +     idx = cachep->memcg_params.id;
> +     VM_BUG_ON(idx == -1);
> +
> +     memcg = mem_cgroup_from_task(current);
> +     if (!mem_cgroup_kmem_enabled(memcg))
> +         return cachep;
> +
> +     if (rcu_access_pointer(memcg->slabs[idx]) == NULL) {
> +         memcg_create_cache_enqueue(memcg, cachep);
> +         return cachep;
> +     }
> +
> +     return rcu_dereference(memcg->slabs[idx]);

```

Is it ok to call rcu_access_pointer() and rcu_dereference() without holding rcu_read_lock()?

```

> + }
> + EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
> +
> + void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
> + {
> +     rcu_assign_pointer(cachep->memcg_params.memcg->slabs[id], NULL);
> + }
> +
> + bool __mem_cgroup_charge_kmem(gfp_t gfp, size_t size)
> + {
> +     struct mem_cgroup *memcg;
> +     bool ret = true;
> +
> +     rcu_read_lock();

```

```

> + memcg = mem_cgroup_from_task(current);
> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> +     goto out;
> +
> + mem_cgroup_get(memcg);

```

Why do we need to get a reference to the memcg for every charge?
How will this work when deleting a memcg?

```

> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> + if (ret)
> +     mem_cgroup_put(memcg);
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_charge_kmem);
> +
> +void __mem_cgroup_uncharge_kmem(size_t size)
> +{
> + struct mem_cgroup *memcg;
> +
> + rcu_read_lock();
> + memcg = mem_cgroup_from_task(current);
> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> +     goto out;
> +
> + mem_cgroup_put(memcg);
> + memcg_uncharge_kmem(memcg, size);
> +out:
> + rcu_read_unlock();
> +}
> +EXPORT_SYMBOL(__mem_cgroup_uncharge_kmem);
> +
> +bool __mem_cgroup_charge_slab(struct kmem_cache *cachep, gfp_t gfp, size_t size)
> +{
> + struct mem_cgroup *memcg;
> + bool ret = true;
> +
> + rcu_read_lock();
> + memcg = cachep->memcg_params.memcg;
> + if (!mem_cgroup_kmem_enabled(memcg))
> +     goto out;
> +
> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> +out:

```

```

> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__mem_cgroup_charge_slab);
> +
> +void __mem_cgroup_uncharge_slab(struct kmem_cache *cachep, size_t size)
> +{
> + struct mem_cgroup *memcg;
> +
> + rcu_read_lock();
> + memcg = cachep->memcg_params.memcg;
> +
> + if (!mem_cgroup_kmem_enabled(memcg)) {
> + rcu_read_unlock();
> + return;
> + }
> + rcu_read_unlock();
> +
> + memcg_uncharge_kmem(memcg, size);
> +}
> +EXPORT_SYMBOL(__mem_cgroup_uncharge_slab);
> +
> +static void memcg_slab_init(struct mem_cgroup *memcg)
> +{
> + int i;
> +
> + for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
> + rcu_assign_pointer(memcg->slabs[i], NULL);
> +}
> #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
> static void drain_all_stock_async(struct mem_cgroup *memcg);
> @@ -4790,7 +5103,11 @@ static struct cftype kmem_cgroup_files[] = {
>
> static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
> {
> - return mem_cgroup_sockets_init(memcg, ss);
> + int ret = mem_cgroup_sockets_init(memcg, ss);
> +
> + if (!ret)
> + memcg_slab_init(memcg);
> + return ret;
> };
>
> static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
> @@ -5805,3 +6122,57 @@ static int __init enable_swap_account(char *s)
> __setup("swapaccount=", enable_swap_account);
>

```

```

> #endif
> +
> + #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
> + int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
> + {
> +     struct res_counter *fail_res;
> +     struct mem_cgroup *_memcg;
> +     int may_oom, ret;
> +     bool nofail = false;
> +
> +     may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
> +         !(gfp & __GFP_NORETRY);
> +
> +     ret = 0;
> +
> +     if (!memcg)
> +         return ret;
> +
> +     _memcg = memcg;
> +     ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> +         &_memcg, may_oom);
> +     if (ret == -ENOMEM)
> +         return ret;
> +     else if ((ret == -EINTR) || (ret && (gfp & __GFP_NOFAIL))) {
> +         nofail = true;
> +         /*
> +          * __mem_cgroup_try_charge() chose to bypass to root due
> +          * to OOM kill or fatal signal.
> +          * Since our only options are to either fail the
> +          * allocation or charge it to this cgroup, force the
> +          * change, going above the limit if needed.
> +          */
> +         res_counter_charge_nofail(&memcg->res, delta, &fail_res);

```

We might need to charge memsw here too.

```

> +     }
> +
> +     if (nofail)
> +         res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
> +     else
> +         ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
> +
> +     if (ret)
> +         res_counter_uncharge(&memcg->res, delta);
> +
> +     return ret;
> + }

```

```
> +
> +void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta)
> +{
> +    if (!memcg)
> +        return;
> +
> +    res_counter_uncharge(&memcg->kmem, delta);
> +    res_counter_uncharge(&memcg->res, delta);
```

Might need to uncharge memsw.

```
> +}
> +#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
> --
> 1.7.7.6
>
```
