
Subject: Re: [PATCH 17/23] kmem controller charge/uncharge infrastructure

Posted by [David Rientjes](#) on Mon, 23 Apr 2012 22:25:59 GMT

[View Forum Message](#) <> [Reply to Message](#)

On Sun, 22 Apr 2012, Glauber Costa wrote:

```
>+/*
>+ * 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]);
>+}
```

```

> +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);

```

This seems horribly inconsistent with memcg charging of user memory since it charges to p->mm->owner and you're charging to p. So a thread attached to a memcg can charge user memory to one memcg while charging slab to another memcg?

```

> +
> + if (!mem_cgroup_kmem_enabled(memcg))
> +     goto out;
> +
> + mem_cgroup_get(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);
```
