
Subject: Re: [PATCH 3/7] per-cgroup slab caches
Posted by [Glauber Costa](#) on Wed, 22 Feb 2012 14:25:46 GMT
[View Forum Message](#) <> [Reply to Message](#)

On 02/22/2012 05:21 AM, KAMEZAWA Hiroyuki wrote:

> On Tue, 21 Feb 2012 15:34:35 +0400

> Glauber Costa<glommer@parallels.com> wrote:

>
>> This patch creates the infrastructure to allow us to register
>> per-memcg slab caches. As an example implementation, I am tracking
>> the dentry cache, but others will follow.
>>
>> I am using an opt-in instead of opt-out system here: this means the
>> cache needs to explicitly register itself to be tracked by memcg.
>> I prefer this approach since:
>>
>> 1) not all caches are big enough to be relevant,
>> 2) most of the relevant ones will involve shrinking in some way,
>> and it would be better be sure they are shrinker-aware
>> 3) some objects, like network sockets, have their very own idea
>> of memory control, that goes beyond the allocation itself.
>>
>> Once registered, allocations made on behalf of a task living
>> on a cgroup will be billed to it. It is a first-touch mechanism,
>> but it follows what we have today, and the cgroup infrastructure
>> itself. No overhead is expected in object allocation: only when
>> slab pages are allocated and freed, any form of billing occurs.
>>
>> The allocation stays billed to a cgroup until it is destroyed.
>> It is kept if a task leaves the cgroup, since it is close to
>> impossible to efficiently map an object to a task - and the tracking
>> is done by pages, which contain multiple objects.
>>
>
> Hmm....can't we do this by
>
> kmem_cache = kmem_cache_create(....,, SLAB_XXX_XXX | SLAB_MEMCG_AWARE)
>
> kmem_cache_alloc(kmem_cache, flags)
> => find a memcg_kmem_cache for the thread. if it doesn't exist, create it.

Where exactly? Apart for the slab flag, I don't follow the rest of your proposal.

>
> BTW, comparing ANON/FILE caches, we'll have many kinds of gfp_t flags.
>
> Do we handle it correctly ?

TODO item =)

>
> Maybe I don't fully understand your implemenatation...but try to comment.

Thanks.

```
>
>
>>
>> +struct memcg_kmem_cache {
>> + struct kmem_cache *cache;
>> + struct mem_cgroup *memcg; /* Should be able to do without this */
>> +};
>> +
>> +struct memcg_cache_struct {
>> + int index;
>> + struct kmem_cache *cache;
>> +};
>> +
>> +enum memcg_cache_indexes {
>> + CACHE_DENTRY,
>> + NR_CACHES,
>> +};
>> +
>> +int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);
>> +void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages);
>> +struct mem_cgroup *memcg_from_shrinker(struct shrinker *s);
>> +
>> +struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index);
>> +void register_memcg_cache(struct memcg_cache_struct *cache);
>> +void memcg_slab_destroy(struct kmem_cache *cache, struct mem_cgroup *memcg);
>> +
>> +struct kmem_cache *
>> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base);
>> +
>>  /* Stats that can be updated by kernel. */
>>  enum mem_cgroup_page_stat_item {
>>   MEMCG_NR_FILE_MAPPED, /* # of pages charged as file rss */
>> diff --git a/include/linux/slab.h b/include/linux/slab.h
>> index 573c809..8a372cd 100644
>> --- a/include/linux/slab.h
>> +++ b/include/linux/slab.h
>> @@ -98,6 +98,14 @@
>>  void __init kmem_cache_init(void);
>>  int slab_is_available(void);
```

```

>>
>> +struct mem_cgroup;
>> +
>> +unsigned long slab_nr_pages(struct kmem_cache *s);
>> +
>> +struct kmem_cache *kmem_cache_create_cg(struct mem_cgroup *memcg,
>> + const char *, size_t, size_t,
>> + unsigned long,
>> + void (*)(void *));
>> struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
>> unsigned long,
>> void (*)(void *));
>> diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
>> index a32bcfd..4f39fff 100644
>> --- a/include/linux/slub_def.h
>> +++ b/include/linux/slub_def.h
>> @@ -100,6 +100,8 @@ struct kmem_cache {
>> struct kobject kobj; /* For sysfs */
>> #endif
>>
>> + struct mem_cgroup *memcg;
>> + struct kmem_cache *parent_slab; /* can be moved out of here as well */
>> #ifdef CONFIG_NUMA
>> /*
>>  * Defragmentation by allocating from a remote node.
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 26fda11..2aa35b0 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -49,6 +49,7 @@
>> #include<linux/page_cgroup.h>
>> #include<linux/cpu.h>
>> #include<linux/oom.h>
>> +#include<linux/slab.h>
>> #include "internal.h"
>> #include<net/sock.h>
>> #include<net/tcp_memcontrol.h>
>> @@ -302,8 +303,11 @@ struct mem_cgroup {
>> #ifdef CONFIG_INET
>> struct tcp_memcontrol tcp_mem;
>> #endif
>> + struct memcg_kmem_cache kmem_cache[NR_CACHES];
>> };
>>
>> +struct memcg_cache_struct *kmem_avail_caches[NR_CACHES];
>> +
>
> What this pointer array holds ? This can be accessed without any locks ?

```

This holds all tracked caches types, with a pointer to the original cache from which the memcg-specific cache is created.

This one can be a list instead, but kmem_cache[] inside the memcg structure, not so sure. I need a quick way to grab the memcg-specific cache from within the cache code (for the shrinkers), therefore I've chosen to do that with the indexes. (Hummm, now thinking, we can just use the root's memcg for that...)

Since every cache has its index in the scheme I implemented, we don't need locking. There is only one writer that always writes a pointer, always to the same location, All the others are readers, and all callers can cope with this being NULL.

```
>
>
>> /* Stuffs for move charges at task migration. */
>> /*
>>  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
>> @@ -4980,6 +4984,93 @@ err_cleanup:
>>
>> }
>>
>> +struct memcg_kmem_cache *memcg_cache_get(struct mem_cgroup *memcg, int index)
>> +{
>> + return &memcg->kmem_cache[index];
>> +}
>> +
>> +void register_memcg_cache(struct memcg_cache_struct *cache)
>> +{
>> + BUG_ON(kmem_avail_caches[cache->index]);
>> +
>> + kmem_avail_caches[cache->index] = cache;
>> +}
>> +
>> +#define memcg_kmem(memcg) \
>> + (memcg->kmem_independent_accounting ? &memcg->kmem : &memcg->res)
>> +
>> +struct kmem_cache *
>> +kmem_cache_dup(struct mem_cgroup *memcg, struct kmem_cache *base)
>> +{
>> + struct kmem_cache *s;
>> + unsigned long pages;
>> + struct res_counter *fail;
>> + /*
>> +  * TODO: We should use an ida-like index here, instead
>> +  * of the kernel address
>> +  */
```

```

>> + char *kname = kasprintf(GFP_KERNEL, "%s-%p", base->name, memcg);
>> +
>> + WARN_ON(mem_cgroup_is_root(memcg));
>> +
>> + if (!kname)
>> +     return NULL;
>> +
>> + s = kmem_cache_create_cg(memcg, kname, base->size,
>> +     base->align, base->flags, base->ctor);
>> + if (WARN_ON(!s))
>> +     goto out;
>> +
>> +
>> + pages = slab_nr_pages(s);
>> +
>> + if (res_counter_charge(memcg_kmem(memcg), pages<< PAGE_SHIFT,&fail)) {
>> +     kmem_cache_destroy(s);
>> +     s = NULL;
>> + }
>
> Why 'pages' should be charged to a new memcg ?
> A newly created memcg starts with res_counter.usage != 0 ??

```

See my last e-mail to Suleiman. It starts with `usage == 0`, unless it creates pages outside the `allocate_slab()` mechanism. For the next version, I'll go read both `slab.c` and `slub.c` extra carefully, and if this is in case impossible, I'll remove this test.

Heck, actually, I will just remove this test no matter what. Even if there are allocations outside of the tracked mechanism, it is easier to track those, than to clutter this here...

```

>> +
>> + mem_cgroup_get(memcg);
>
> get even if allocation failure ? (and for what updating refcnt ?)

```

No, if we reached this point, the allocation succeeded. We don't need to keep the memcg structure alive itself, but we need to keep the cache alive (since it will still have objects upon memcg destruction).

As a policy, I am trying to avoid putting more stuff in struct `kmem_cache`, so I put it here.

```

>
>
>> +out:
>> + kfree(kname);

```

```

>> + return s;
>> +}
>
>
>> +
>> +int memcg_kmem_newpage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
>> +{
>> + unsigned long size = pages<< PAGE_SHIFT;
>> + struct res_counter *fail;
>> +
>> + return res_counter_charge(memcg_kmem(memcg), size,&fail);
>> +}
>> +
>> +void memcg_kmem_freepage(struct mem_cgroup *memcg, struct page *page, unsigned long
pages)
>> +{
>> + unsigned long size = pages<< PAGE_SHIFT;
>> +
>> + res_counter_uncharge(memcg_kmem(memcg), size);
>> +}
>> +
>> +void memcg_create_kmem_caches(struct mem_cgroup *memcg)
>> +{
>> + int i;
>> +
>> + for (i = 0; i< NR_CACHES; i++) {
>> + struct kmem_cache *cache;
>> +
>> + if (!kmem_avail_caches[i] || !kmem_avail_caches[i]->cache)
>> + continue;
>> +
>> + cache = kmem_avail_caches[i]->cache;
>> +
>> + if (mem_cgroup_is_root(memcg))
>> + memcg->kmem_cache[i].cache = cache;
>> + else
>> + memcg->kmem_cache[i].cache = kmem_cache_dup(memcg, cache);
>> + memcg->kmem_cache[i].memcg = memcg;
>> + }
>> +}
>
> Hmm... memcg should know _all_ kmem_caches when it's created. Right ?
> Then, modules will not use memcg aware kmem_cache...

```

Which maybe is not a bad thing. Again, see my last e-mail to Suleiman, I described my reasons there.

But thing is, we can always reclaim user pages. But this is not true for kernel pages. We need the help of a shrinker. So if you don't have a memcg-aware shrinker, maybe it would be better not to get tracked in the first place.

```
>> +
>> +
>> static struct cgroup_subsys_state * __ref
>> mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup *cont)
>> {
>> @@ -5039,6 +5130,8 @@ mem_cgroup_create(struct cgroup_subsys *ss, struct cgroup
>> *cont)
>> if (parent)
>>     memcg->swappiness = mem_cgroup_swappiness(parent);
>> atomic_set(&memcg->refcnt, 1);
>> +
>> + memcg_create_kmem_caches(memcg);
>> memcg->move_charge_at_immigrate = 0;
>> mutex_init(&memcg->thresholds_lock);
>> return &memcg->css;
>> diff --git a/mm/slub.c b/mm/slub.c
>> index 4907563..f3815ec 100644
>> --- a/mm/slub.c
>> +++ b/mm/slub.c
>> @@ -31,6 +31,8 @@
>> #include<linux/stacktrace.h>
>>
>> #include<trace/events/kmem.h>
>> +#include<linux/cgroup.h>
>> +#include<linux/memcontrol.h>
>>
>> /*
>>  * Lock order:
>> @@ -1281,6 +1283,7 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t flags,
>> int node)
>>     struct page *page;
>>     struct kmem_cache_order_objects oo = s->oo;
>>     gfp_t alloc_gfp;
>> + int pages;
>>
>>     flags&= gfp_allowed_mask;
>>
>> @@ -1314,9 +1317,17 @@ static struct page *allocate_slab(struct kmem_cache *s, gfp_t
>> flags, int node)
>>     if (!page)
>>         return NULL;
>>
>> + pages = 1<< oo_order(oo);
```

```
>> +
>> + #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
>> + if (s->memcg && memcg_kmem_newpage(s->memcg, page, pages) < 0) {
>> +   __free_pages(page, oo_order(oo));
>> +   return NULL;
>> + }
>> + #endif
>
> Hmm. no reclaim happens at allocation failure ?
> Doesn't it turn to be very bad user experience ?
```

Yes, I'll handle it in the next respin.
