Subject: Re: [PATCH v5 06/14] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 18 Oct 2012 09:23:28 GMT
View Forum Message <> Reply to Message

On 10/18/2012 02:37 AM, David Rientjes wrote:
> On Tue, 16 Oct 2012, Glauber Costa wrote:
>
>> + /* If the test is dying, just let it go. */
>> + if (unlikely(test_thread_flag(TIF_MEMDIE)
>> + || fatal_signal_pending(current)))
>> + return true;
>
> This can be simplified to just check fatal_signal_pending(), all threads
> with TIF_MEMDIE also have a pending SIGKILL.

Yes, I believe it is better. I will change.

>> +
>> + return __memcg_kmem_newpage_charge(gfp, memcg, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_uncharge_page: uncharge pages from memcg
>
> Should be memcg_kmem_uncharge_pages() since it takes an order argument?
>

I tried to use naming as close as possible to user-memcg. But to be
fair, their are always calling it page-by-page, so pages() won't be a
problem here.

>> + * @page: pointer to struct page being freed
>> + * @order: allocation order.
>> + *
>> + * there is no need to specify memcg here, since it is embedded in page_cgroup
>> + */
>> +static __always_inline void
>> +memcg_kmem_uncharge_page(struct page *page, int order)
>> +{
>> + if (memcg_kmem_enabled())
>> + __memcg_kmem_uncharge_page(page, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_commit_charge: embeds correct memcg in a page
>> + * @page: pointer to struct page recently allocated
>> + * @memcg: the memcg structure we charged against
>> + * @order: allocation order.

>> + *
>> + * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
>> + * failure of the allocation. if @page is NULL, this function will revert the
>> + * charges. Otherwise, it will commit the memcg given by @memcg to the
>> + * corresponding page_cgroup.
>> + */
>> +static __always_inline void
>> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
>> +{
>> + if (memcg_kmem_enabled() && memcg)
>> + __memcg_kmem_commit_charge(page, memcg, order);
>> +}
>> +
>> #else
>> static inline void sock_update_memcg(struct sock *sk)
>> {
>> @@ -406,6 +489,21 @@ static inline void sock_update_memcg(struct sock *sk)
>> static inline void sock_release_memcg(struct sock *sk)
>> {
>> }
>> +
>> +static inline bool
>> +memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
>> +{
>> + return true;
>> +}
>> +
>> +static inline void memcg_kmem_uncharge_page(struct page *page, int order)
>
> Two spaces.
>

Thanks.

>> +{
>> +}
>> +
>> +static inline void
>> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
>> +{
>> +}
>> #endif /* CONFIG_MEMCG_KMEM */
>> #endif /* _LINUX_MEMCONTROL_H */
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 30eafeb..1182188 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c

>> @@ -10,6 +10,10 @@
>>   * Copyright (C) 2009 Nokia Corporation
>>   * Author: Kirill A. Shutemov
>>   *
>> + * 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
>> @@ -2630,6 +2634,171 @@ static void __mem_cgroup_commit_charge(struct mem_cgroup
*memcg,
>>   memcg_check_events(memcg, page);
>> }
>>
>> +#ifdef CONFIG_MEMCG_KMEM
>> +static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
>> +{
>> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
>> + (memcg->kmem_accounted & KMEM_ACCOUNTED_MASK);
>> +}
>> +
>> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
>> +{
>> + struct res_counter *fail_res;
>> + struct mem_cgroup *_memcg;
>> + int ret = 0;
>> + bool may_oom;
>> +
>> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
>> + if (ret)
>> + return ret;
>> +
>> + /*
>> + * Conditions under which we can wait for the oom_killer.
>> + * We have to be able to wait, but also, if we can't retry,
>> + * we obviously shouldn't go mess with oom.
>> + */
>> + may_oom = (gfp & __GFP_WAIT) && !(gfp & __GFP_NORETRY);
>
> What about gfp & __GFP_FS?
>

Do you intend to prevent or allow OOM under that flag? I personally
think that anything that accepts to be OOM-killed should have GFP_WAIT
set, so that ought to be enough.

```
>> +
>> + _memcg = memcg;
>> + ret = __mem_cgroup_try_charge(NULL, gfp, size >> PAGE_SHIFT,
>> +         &_memcg, may_oom);
>> +
>> + if (ret == -EINTR)  {
>> +  /*
>> +   * __mem_cgroup_try_charge() chosed 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, do
>> +   * it as a temporary condition. But we can't fail. From a
>> +   * kmem/slab perspective, the cache has already been selected,
>> +   * by mem_cgroup_get_kmem_cache(), so it is too late to change
>> +   * our minds. This condition will only trigger if the task
>> +   * entered memcg_charge_kmem in a sane state, but was
>> +   * OOM-killed.  during __mem_cgroup_try_charge. Tasks that are
>
> Looks like some copy-and-paste damage.
>
```

thanks.

```
>> +void __memcg_kmem_uncharge_page(struct page *page, int order)
>> +{
>> + struct mem_cgroup *memcg = NULL;
>> + struct page_cgroup *pc;
>> +
>> +
>> + pc = lookup_page_cgroup(page);
>> + /*
>> +  * Fast unlocked return. Theoretically might have changed, have to
>> +  * check again after locking.
>> +  */
>> + if (!PageCgroupUsed(pc))
>> +  return;
>> +
>> + lock_page_cgroup(pc);
>> + if (PageCgroupUsed(pc)) {
>> +  memcg = pc->mem_cgroup;
>> +  ClearPageCgroupUsed(pc);
>> + }
>> + unlock_page_cgroup(pc);
>> +
>> + /*
>> +  * We trust that only if there is a memcg associated with the page, it
>> +  * is a valid allocation
>> +  */
>> + if (!memcg)
```

>> + return;
>> +
>> + VM_BUG_ON(mem_cgroup_is_root(memcg));
>> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
>> + mem_cgroup_put(memcg);
>
> Should this mem_cgroup_put() be done conditionally on
> memcg->kmem_accounted & KMEM_ACCOUNTED_MASK?
>
> The next patch in the series does memcg_kmem_newpage_charge() in the page
> allocator which will return true for memcg_can_account_kmem() without
> doing mem_cgroup_get().
>

And then this put will go away as well.

I am not testing for memcg_can_account_kmem in here, because having or not having the PageCgroupUsed bit set (and therefore, a valid memcg) in page_cgroup should be the most robust test here.
